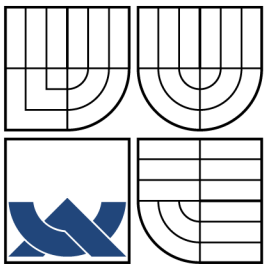
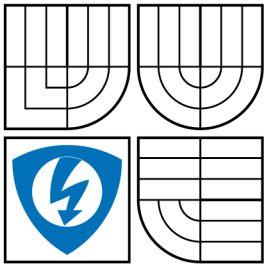


BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ



FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION
DEPT. OF CONTROL AND INSTRUMENTATION

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ
ÚSTAV AUTOMATIZACE A MĚŘICÍ TECHNIKY

DESIGN AND IMPLEMENTATION OF CONTROL SOFTWARE LIBRARIES FOR FIBER CHARACTERIZATION

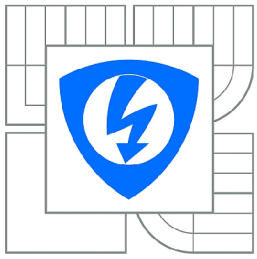
DESIGN AND IMPLEMENTATION OF CONTROL SOFTWARE LIBRARIES FOR FIBER CHARACTERIZATION

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR
AUTOR PRÁCE
Bc. LADISLAV PODIVÍN

SUPERVISOR
VEDOUCÍ PRÁCE
Ing. PETR HONZÍK , Ph.D.

BRNO 2010



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta elektrotechniky a komunikačních technologií

Ústav automatizace a měřicí techniky

Diplomová práce

magisterský navazující studijní obor
Kybernetika, automatizace a měření

Student: Bc. Ladislav Podivín ***ID:*** 78309
Ročník: 2 ***Akademický rok:*** 2009/2010

NÁZEV TÉMATU:

Design and implementation of control software libraries for fiber characterization

POKYNY PRO VYPRACOVÁNÍ:

1 Design and implementation of real-time image processing library with C++
- In this topic a real-time image processing library for Linux will be developed. The design is made in co-operation with another MSc. student.
- The library will be developed on real-time environment and shall employ Qt4 framework
- Selection of suitable real-time kernel is part of the topic.
-The goal is to develop a platform, which has sufficient processing capabilities for machine vision applications and allows flexible changing of used machine vision algorithms.

2 Design and implementation of Haptic interface with force-feedback
- This topic includes design and implementation for a library controlling a haptic device, Phantom Desktop (SensAble).
- The haptic device receives its feedback from a force sensor or from a machine vision library.
- The goal is to develop an application, which can be integrated into the mentioned framework. The given interfaces must be used without changes.

DOPORUČENÁ LITERATURA:

Termín zadání: 8.2.2010 ***Termín odevzdání:*** 2.8.2010

Vedoucí práce: Ing. Petr Honzík, Ph.D.

prof. Ing. Pavel Jura, CSc.
Předseda oborové rady

UPOZORNĚNÍ:

Autor diplomové práce nesmí při vytváření diplomové práce porušit autorská práva třetích osob, zejména nesmí zasahovat nedovoleným způsobem do cizích autorských práv osobnostních a musí si být plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č.40/2009 Sb.

ABSTRACT

This thesis deals with the design and the implementation of two software libraries (often referred as *modules* in the following text). The modules are parts of the distributed control system *CoSMic*, which is meant to control a special hardware platform for paper fiber characterization.
The first of the two modules is *HapticFiber* - module to provide an interface between the control system and a haptic input device. The second one is *ViCo* - module to create a software envelop to hold a user defined image processing algorithm. This module must be ready to fulfill certain time restrictions, that is why it needs to be run on a real-time operating system.

KEYWORDS

haptic device, machine vision, real-time operating system, control system

ABSTRAKT

Tato práce se zabývá návrhem a implementací dvou konkrétních softwarových modulů, které jsou částí distribuovaného řídícího systému *CoSMic*. Tento systém je určen pro řízení speciálního zařízení pro charakterizaci papírových vláken.
Prvním vyvinutým modulem je *HapticFiber*, ten má poskytovat rozhraní mezi řídícím systémem a speciálním vstupním zařízením - *haptic device*. Druhým modulem je *ViCo*, jehož účelem je poskytnout softwarovou obálku pro uživatelem definovaný algoritmus zpracovaní obrazu. Tento modul musí být připraven splnit určitá časová omezení, proto je nutné, aby běžel v rámci operačním systému reálného času.

KLÍČOVÁ SLOVA

haptic device, počítačové vidění, real-time operační systém, řídicí systém

PODIVÍN, Ladislav *Design and implementation of control software libraries for fiber characterization*: master's thesis. Brno: Brno University of Technology, Faculty of Electrical Engineering and Communication, Dept. of Control and Instrumentation, 2010. 71 p. Supervised by Ing. Petr Honzík , Ph.D.

DECLARATION

I declare that I have elaborated my master's thesis on the theme of "Design and implementation of control software libraries for fiber characterization" independently, under the supervision of the master's thesis supervisor and with the use of technical literature and other sources of information which are all quoted in the thesis and detailed in the list of literature at the end of the thesis.

As the author of the master's thesis I furthermore declare that, concerning the creation of this master's thesis, master's thesis, I have not infringed any copyright. In particular, I have not unlawfully encroached on anyone's personal copyright and I am fully aware of the consequences in the case of breaking Regulation § 11 and the following of the Copyright Act No 121/2000 Vol., including the possible consequences of criminal law resulted from Regulation § 152 of Criminal Act No 140/1961 Vol.

Brno ....

(author's signature)

I would like to use this space to thank all the members of the Micro- and Nanosystems Research Group for creating such a great working environment. I especially appreciate the trust Prof. Pasi Kallio gave me, and all the support from my colleague Mathias von Essen.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

This thesis describes a software project which is a part of the Control Software for Microrobotic Platform (CoSMic) framework. CoSMic is a distributed system designed to control a hardware microrobotic platform for paper fiber characterization. The platform consists of three main subsystems - microrobotic actuators, machine vision and data acquisition system. CoSMic is supposed to control each of the subsystems platform fully autonomously, but currently this requirement cannot be satisfied, because of missing software modules. CoSMic consists of two main software parts - Control of Micromanipulation System and Control of Data Acquisition System. It is necessary to extend CoSMic's control capabilities by implementing new modules. To develop and implement two of them is the goal of this thesis.

All the software described in the thesis was developed in C++ language and highly relies on *Qt* framework, introduced in Section 2.3. The *QtCreator* [24] was used as Integrated Development Environment (IDE).

All the development was performed in the Micro and Nanosystems Research Group of the Department of Automation Science and Engineering at Tampere University of Technology in Finland.

1.1 Role of This Thesis

The main goal of this thesis is to design and implement two new parts of the CoSMic framework according to given requirements. The first part is the Haptic Module. Its purpose is to create a layer connecting a haptic device with the framework. A haptic device could be shortly characterized as a tactile input output device using force feedback to create virtual objects [2]. The particular haptic device used in this project was the *Sensable PHANTOM Desktop* [3], which is in fact a force feedback joystick with six degrees of freedom. The software layer should be responsible especially for the safe data exchange with the device.

The other part is the Machine Vision part - also called Vision System Control (ViCo). Its purpose is to create a software envelop for user defined image processing algorithm. Thanks to such an envelop a user does not need to care about camera

data acquisition, threading and other problems. His/her responsibility is to define the desired image processing algorithm. For a better idea what ”define” means in this case, see Section 4.2. For a better idea, what it could mean in the future, see 6.2.

Moreover, the envelop is supposed to meet some processing deadlines - i.e. to keep processing images in certain bounded time. The particular time requirements depend on a chosen image processing algorithm, so they cannot be specified now, but the envelop has to be ready to satisfy them. This implies that the other important goal of the thesis is to choose a suitable real time operating system.

1.2 Thesis Structure

Chapter 2 is dedicated to a description of used software frameworks and especially to a survey of suitable real time operating systems. Chapters 3 and 4 both deal with software design - the former is about the Haptic Module, while the latter about the Machine Vision Module. Chapter 5 introduces results of the tests taken to verify the implemented modules. Finally, Chapter 6 provides overall conclusion and lists steps needed to be done as a part of a future development.

Appendix A provides a few hints on how to speed up Linux kernel compilation and Appendix B summarizes versions of all used software.

Another important supplement of the thesis is the source code documentation generated with *Doxygen* tool [4]. The documentation is a standalone source of detailed information about particular classes and methods. Although the documentation is meant to be independent on the thesis, references to it are frequent in the text. That is the way to connect the abstract approaches described with their real implementation. To make this connection easier, the documentation provides a full-text search.

2 SOFTWARE BACKGROUND

This chapter describes software projects the developed modules are based on. The first section describes suitable real time operating system. The next one chooses the best system for the Machine Vision Module. The rest of the chapter is dedicated to the important *Qt* framework and the very last part briefly describes the *OpenCV* framework.

2.1 Suitable Real-Time Operating Systems

An operating system is a piece of software responsible for computer's resources management and providing an environment for other programs to run. In fact it is an abstract layer between user applications and hardware.

Real Time Operating System (RTOS) is a system which is always able to meet certain deadlines [5][6]. For example, if an event arises (e.g. external interrupt) the maximal time needed to serve it must be bounded. In other words, the maximal time needed to perform every operation can be determined.

The next question is, how to achieve such behavior. The system has to be based on preemptive multitasking, so its essential part is a deterministic scheduler with system of priorities [7]. Such system also has to handle interrupts in deterministic way. Anyway, this is just a brief description provided for better understanding of the following text, a comprehensive description is out of the scope of this work.

Although all developers take care of a maximal code portability, the preferred operating system for the CoSMic platform is Linux. So Linux compatibility of a chosen Real Time (RT) solution is the requirement number one. Linux itself is not an RTOS, but there are several extensions enabling it to run RT tasks. Some of those systems are for free, which leads us to the second requirement - the price.

The main advantage of commercial solutions over the free ones is obviously a better customer support. In case of the free systems, one can rely only on his own skills and especially on help from community. However, I decided not to burden

the project budged, to take the risk and to try one of the non-commercial systems. This decision was quite easy, because the CoSMic is not intended to control any potentially dangerous technological processes. In this situation we can just try a free operating system and abandon it in the future, if it does not work properly, because it is always less painful to replace a cost free solution with a commercial one than vice-versa.

The possible Linux based and cost free systems are Linux with RT patches, Xenomai, RTAI and RTLinuxFree. Next subsections are dedicated to description of the first three of them. The RTLinuxFree is not discussed here, because it is just a side project of Wind River company developing the commercial RTLinux. The commercial version has most likely a bigger priority for the company and moreover the community around the free version is quite small. It is impossible to prove the first statement without buying the commercial license. It is quite difficult to prove the second statement too, because one cannot blindly rely on an advice of people from related internet discussions. At least a small proof is that Google returns approximately 1000 results concerning word RTLinuxFree, and circa 55000 for Xenomai.

Even if all this RTLinuxFree investigation was wrong, there are still three more systems to choose from, which decreases a possible impact of a bad decision.

Remark on System Management Interrupts

Since the CoSMic is intended to run on x86 or x86_64 architectures, the System Management Interrupt (SMI) has to be mentioned, before we can proceed to the next sections describing particular systems.

The SMI is a hardware interrupt of the highest priority switching Central Processing Unit (CPU) into the System Management Mode (SMM). In this mode all execution is suspended and a special software takes control [8]. This can be unpleasant even if one does not use an RTOS - e.g. buggy firmware can omit a clean up and let a hardware in an undefined state. In case, one uses an RTOS, the SMIs may

become very good source of high latencies, because a RT task of high priority can be interrupted by the SMI for unspecified amount of time and an operating system cannot do anything about that.

SMIs used to be just a part of power management, but nowadays they cover a wider range - e.g. they emulate missing or buggy hardware [8]. There are usually ways to disable SMIs but it may lead to hardware malfunction or in extreme cases even to hardware damage (but this is said to be rare).

Anyway, the SMI is an Intel specific issue. On the other hand, Section 2.4 shows, that Intel processors have certain advantage for this project and moreover the previous sentence does not say, that all Intel processors mean a serious problem with the SMI - it depends also on a chipset etc.

2.1.1 Linux with CONFIG_PREEMPT_RT Patch

Standard Linux 2.6 series kernel can be compiled to be almost fully preemptible. Documentation of the related compilation option `CONFIG_PREEMPT` says: *"This option reduces the latency of the kernel by making all kernel code (that is not executing in a critical section) preemptible."* [9]

The patch `CONFIG_PREEMPT_RT` changes Linux kernel to be fully preemptible. Since fully preemptible kernel is not sufficient guarantee of RT behavior, the patch adds also few more features [10]. There will be mentioned only the most important ones here.

The first significant change introduced by the patch is, that all interrupt service routines run in dedicated threads, thus allowing to be preempted. Another important feature is adding the priority inheritance to synchronization objects to prevent the priority inversion problem.

To apply this patch and compile the kernel is most likely the easiest way to get a working RTOS. Moreover, if an existing Linux application was implemented with the RT manners in mind, it can run in RT without a recompilation.

Linux with `CONFIG_PREEMPT_RT` patch provides no official SMI workaround.

2.1.2 Xenomai

The Xenomai project introduces an approach of two kernels running beside each other. One is the normal Linux kernel and the other is Xenomai *nucleus*. The project called ADEOS ensures this symbiosis. Its purpose is to provide a layer enabling hardware sharing among multiple operating systems [11].

The most important part of ADEOS is the way it distributes events (mainly interrupts). It creates a pipeline (called *i-pipe*) providing a possibility to process events sequentially by all the operating systems. Once an interrupt is served by the first operating system, it is passed to the next one and then to the next one etc. Xenomai kernel is in the pipeline before Linux kernel, which ensures all interrupts are delivered to the real time kernel first. Xenomai may also ask the i-pipe not to pass interrupts to the Linux kernel.

Xenomai allows to create RT tasks either in kernel or user space. The kernel tasks are considered deprecated [12] and their support will be discontinued in the future. Anyway, there are two modes of execution for the user space tasks. Normally a RT task runs in Xenomai domain (*primary execution mode*), but it can also call Linux services. Every call to a Linux system service causes the task to be switched to the Linux domain (*secondary execution mode*).

A task in the primary mode is scheduled by Xenomai nucleus and by Linux kernel while in the secondary mode. As discussed already, Linux is not an RTOS, so this switch may violate RT behavior of the task. Anyway, task's priority remains the same no matter of the mode.

If Xenomai is configured to do so, it makes the ADEOS interrupt pipeline to block interrupt propagation towards the Linux kernel while a Xenomai task is performing a Linux system call (so called *interrupt shield*). This prevents the task from being preempted by any Linux interrupt service routine. All interrupts are delivered after the blocking is over.

An important information is, that a task leaves the secondary mode when it calls Xenomai system call that requires the switch [13] - not any sooner.

It is also possible to create a standard Linux task (thus running in the secondary mode from Xenomai point of view) and turn it into a Xenomai task.

To provide more flexibility there are several skins stacked over the nucleus. Each skin represents one Application Program Interface (API), which makes a chameleon system from Xenomai. The purpose of the skins is to ease porting of existing applications to other operating systems. Refer to the documentation for full list of supported APIs. Xenomai provides skin called *native*, which is the preferred skin if one does not need the API compatibility with any other operating system. Anyway, this native skin has no privileged status among the others [13].

As the described task migration between the two kernels indicates, Xenomai is also very closely integrated to the Linux environment. It is possible for Xenomai tasks to receive Linux signals, to be debugged with standard Linux debugger *gdb* etc. [14].

Xenomai supports the Real-Time Driver Model (RTDM), which unifies environments for real time Linux drivers [15]. Another important part of Xenomai called *Analogy* is based on RTDM. Analogy is intended to support data acquisition hardware under Xenomai. Unfortunately, at the time of writing this thesis the current Analogy implementation (Xenomai 2.5.3) provides only the very basic functionality. Important functions concerning a hardware calibration are still missing.

Xenomai provides a detection of chipsets that use the SMI and also a possibility to disable either all SMIs or just selected ones.

2.1.3 RTAI

The basic idea of Real Time Application Interface (RTAI) is very similar to Xenomai, it is also based on ”two kernels” approach using ADEOS. Although ADEOS is the main common part, the main difference lies in the way it is used. Unlike Xenomai Rtai uses so called immediate interrupt dispatching. In case of Xenomai, interrupts are delivered via ADEOS first to the Xenomai nucleus and then to the Linux kernel. Rtai bypasses ADEOS and takes the interrupts directly and then lets ADEOS to

pass them to Linux kernel [16]. This leads logically to a better performance of RTAI tasks, since this approach skips one layer. On the other hand, it is obvious that such a layer skipping is not the cleanest solution.

Another significant difference to Xenomai is, that RTAI provides only one API. However, it also supports both kernel and user space tasks as well as the RTDM.

The situation about primary and secondary mode described in the Xenomai section is also very similar here, although RTAI developers do not use this primary/secondary terminology. Any Linux system call puts a RTAI task under control of Linux kernel until the call is done and a RTAI system call is called. Unlike Xenomai, RTAI can be configured to force switching back to RTAI domain immediately after finishing the Linux system call - i.e. without the need of a following RTAI system call [17].

The main advantage RTAI has over Xenomai is the RTAI-Lab project. It is a set of tools allowing to convert block diagrams created in Matlab/Simulink or Scilab/Scicos to RTAI executable and tools to interact with the the running executable [18]. Xenomai does not provide any similar utilities.

Although to generate a code from Simulink block diagrams could be useful in the future (as discussed in Section 4.2.2), this advantage is not that significant, because there are also other ways of the code generating and the possible interconnection between Matlab and running binary is not necessary.

RTAI also supports *Comedi* framework. Comedi is a set of drivers and libraries to work with data acquisition hardware under Linux [19]. By ”supports” is meant, that it is possible to use Comedi from kernel space and user space tasks without violation of the RT behavior. Obviously, it is needed to use a special API instead of the one normally available on Linux [19].

RTAI provides similar services concerning the SMI detection and disabling, as Xenomai does.

2.2 Comparison and Conclusion

Xenomai and RTAI used to be one project called *RTAI/Fusion*. Their structure is quite similar. According to Xenomai developers the main difference lies in the goals of the projects [20]. They say, RTAI is focused on maximal performance while Xenomai pays more attention to the portability (mainly the mentioned skins) and code quality, thus maintainability.

Xenomai supports more hardware architectures than RTAI [21] and its configuration is more user friendly, since it tries to get kernel patching more automated and its main configuration is embedded into the Linux kernel one. On the other hand RTAI has better support for RT data acquisition.

After reading of Section 2.1.1 dealing with the `CONFIG_PREEMPT_RT` patch, one can possibly ask - why is there a need for Xenomai, RTAI and others when Linux itself can be converted to an RTOS?. Xenomai and RTAI were first released in the times when the RT patch was very young and not providing full RT behavior. Another question is, how the today's situation is. Unfortunately the answer is rather tricky. There is a lack of studies comparing capabilities of those systems and developers of Xenomai as well as developers of RTAI are very diplomatic while trying to answer this question - in fact they say "try it and you will see". Anyway, the fact the projects RTAI and Xenomai still exist and there are still people using them implies the RT patch solution is still not perfect, although there is no real proof.

I decided to use Xenomai, because it has a clear vision of its future development. In case of the future release (Xenomai 3 series) a user will be given a choice to use the described "two-kernel" approach or to combine Xenomai with the RT patched Linux kernel [22]. The latter means to use the RT capabilities the patched kernel provides and to add the useful Xenomia skins, the SMI workaround etc. Moreover, it will be still possible to use the "two-kernel" approach in situations where it gets better results (e.g. architectures better supported by original Xenomai).

So to choose Xenomai means, in fact, to choose the RT patched kernel approach

too. Logically, if one takes two from three possibilities, the probability the decision is right, is bigger than if only one approach is selected.

2.3 Qt Framework

"Qt is a cross-platform application and UI framework" [23]. It is a vast set of general purpose C++ classes similar to the basic C++ Standard Template Library (STL). Unlike the STL, a significant part of the framework is dedicated to a Graphical User Interface (GUI) creation support.

Qt supports all the major operating systems like Windows, Linux Mac OS X and few others. As stated, Qt is C++ based, but supports a few other languages (Python, Java, Ruby etc.) through bindings. Thanks to all those facts, all CoSMic software modules are based on Qt - including the two modules discussed in this thesis.

An interesting part of Qt is the threading support. It provides classes encapsulating platform dependent threading routines, thus providing platform independent threading. Those classes for Linux/Unix environments are based on Portable Operating System Interface (POSIX) threads, so it is possible to use the Qt threading even in POSIX compliant RTOSes.

Qt also provides development tools including the IDE *Qt Creator* [24].

2.3.1 Signals and Slots

This mechanism is probably the biggest advantage which Qt brings. It is intended to replace the traditional callback approach, which is not flexible enough and not type safe [25].

Instead of callbacks, Qt uses pairs of *signals* and *slots*. An object can emit a signal to let other objects know about any change of its state. If this signal is connected to a slot of any other object (or even the same object), the slot is executed. A slot is a normal method and thus can be called also in the usual way.

A great benefit is, that it is possible to connect multiple signals to one slot or one signal to many slots and even to connect a signal to another signal or signals.

It is possible to transfer signals across threads, but one must be aware of certain rules [25].

It is obvious that signals and slots are more flexible techniques than the traditional callbacks. On the other hand, signals and slots are little bit slower than callbacks thanks to certain background operations. Callbacks are quick, because all, that must be done, is just to call a function one have a pointer to.

Another drawback is, that template classes cannot use the signal oriented approach [26].

2.3.2 Meta-Object Compiler

The described signals and slots are extensions to the standard C++ [27]. Since programs using Qt are intended to use standard compilers, there is a need for a tool to convert those extensions into the standard C++. Qt uses a mechanism of meta-objects generated with the meta-object compiler [28] from a source code containing the extensions.

2.4 OpenCV

OpenCV stands for Open Source Computer Vision and it is a free image processing software library written in C++ [29]. It does not provide only image processing functionality, it offers also some other services - e.g. classes to represent a generic input device or a generic video writer. This effort should enable user to focus mainly on the image processing.

OpenCV's performance can be boosted using Intel Integrated Performance Primitives (IPP) framework, which is a set of optimized routines for multimedia data processing [30]. This framework is available for Intel processor only.

3 HAPTIC MODULE SOFTWARE DESIGN

A haptic device is an input output device providing an interface between computer and human. It is the input device, because it behaves as a tactile position sensor usually sensing a position of its handle. It is the output device, because the handle is capable of generating forces and thus move itself or prevent a user from moving it. Thanks to these abilities a haptic device is suitable for creating virtual objects, an operator can touch [2].

This module is not supposed to generate any virtual object. This is a task for framework structures built in top of this module. The haptic module should provide a layer between the *3DTouch* library and the rest of the framework. The library is shipped together with the *Sensable Phantom Desktop* device and defines certain low level functionality to control the device.

This library is callback oriented. It controls the physical device from a special thread and a user of the library is allowed to communicate with the device via callback functions. From the software point of view, the most important objective of the module is to implement suitable callbacks and to provide a comfortable callback management. The main part of the management is implemented as a state machine, switching suitable callbacks according to the situation.

Figure 3.1 shows the main module's classes and their relationships. The most important class is the `CHaptCore` which provides the mentioned callback management functionality. The class `CHapticFiber` implements particular callbacks and it is meant to be instantiated in a program using this module. The class `ChapticHW` encapsulates the low-level device related functionality. The classes `CHapticOut` and `CHapticIn` are responsible for the data exchange between a haptic device and another software module. This topic is discussed in Section 3.1 and Section 3.2. The `CAxisLocker` is responsible for the approach of the axis locking described in Section 3.2.

This module is meant to be executed in the Windows environment. To be more specific - Windows XP and better. Anyway, at cost of minor project file modifica-

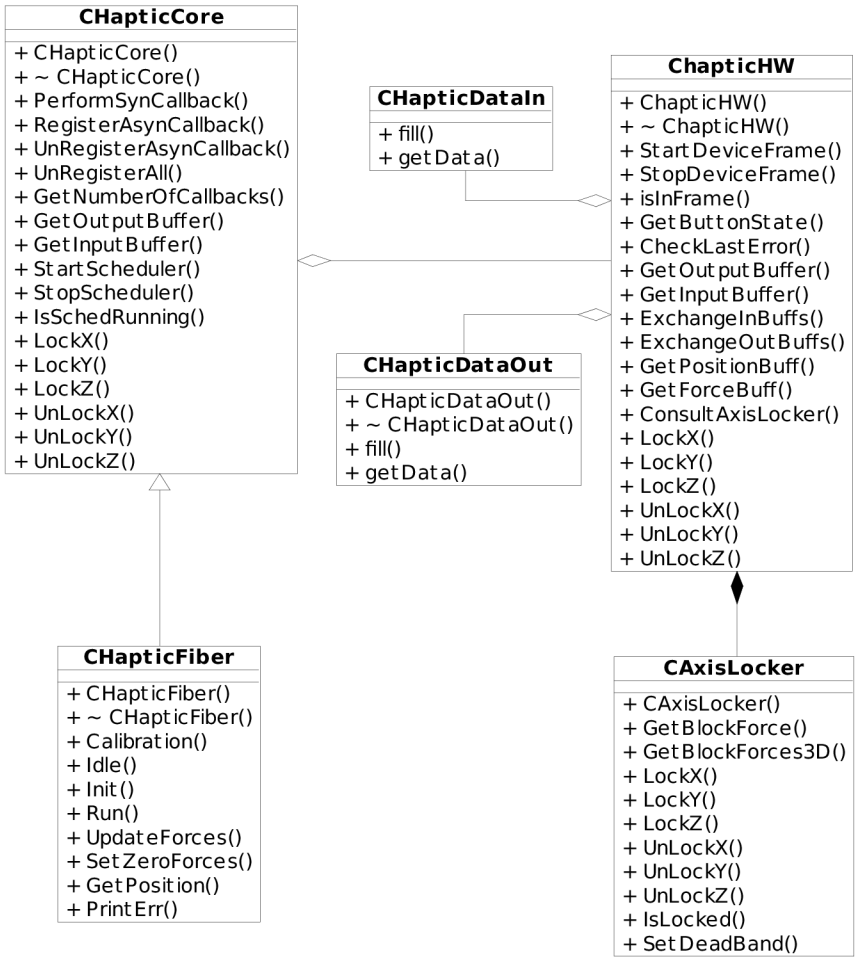


Fig. 3.1: Haptic module class diagram - main classes

tions, it can be compiled for the Linux environment too.

To summarize the above run environment discussion - there are no real time requirements for the Haptic Module.

In the Introduction section it was mentioned, that the main goal of this module is to exchange data with the rest of the framework. Let us focus on the more detailed description of the goal. To exchange data with the device means to grab information about the device handle coordinates and to send values of desired forces, the device is supposed to generate. The next two subsections are dedicated to these topics, while the last subsection covers the topic of the software state machine.

3.1 Device Output Processing

The Sensable Phantom Desktop haptic device has six degrees of freedom - three Cartesian axes plus three angles (roll, pitch and yaw). Currently, the module asks the device only for the 3D Cartesian information, the angles are ignored. It is possible to get the full position information after minor code changes. This is described in the source code documentation in Section "Data Exchange".

The device is asked to provide the handle position as 3D Cartesian coordinates in the device space and the unit of measurement is millimeter. The device space is fixed and its consistency is ensured by the calibration procedure. The output values are meant to control positioning actuators working in their own Cartesian coordinate system, so the Haptic Module needs to convert the data from one space to another. The first idea was to directly map the device output to the actuator space via a linear transformation, but this approach has a serious disadvantage. The assumed movement of the actuators is much slower compared to the handle movement, which depends only on user's will. The question is, how to deal with the situation when the user takes a sequence of quick moves in various directions. The controlled actuators can either follow the drawn trajectory or just track the last point of it.

The first possibility is not flexible enough, because when the actuators would follow the way-points, they logically cannot respond to actual data quick enough, which can be even dangerous in certain situations.

The second approach is not suitable when a user changes the point to follow frequently, because that means to keep sending to an actuator requests for the new position immediately followed by requests to stop the movement. Moreover, there is no guarantee that all actuator types provide the command to interrupt current movement.

The solution of this problem is not to map device's coordinates directly to actuator's coordinates, but to map handle displacement to the desired speed of the actuators. In fact, this means to simulate behavior similar to the well known joystick controller. When the handle is in its zero position, nothing is happening, but when

it is displaced, the desired speed is set to be proportional to the displacement. So this linear conversion (in fact scaling) is another task for this module.

The used device does not provide any built-in scaling of the output. That is why the Haptic Module provides functionality for linear output scaling according to a defined range of desired output values. The real range of device's output values is mapped linearly to the desired range. The real range of device's output, which is in fact the boundary of the device space, is asked directly from the device.

Moreover, a dead band can be defined. The dead band is a zone around handle's zero point where any displacement is considered as zero. The purpose of this measure is to minimize an influence of disturbing effects caused by a human operator - for example shaking hands.

The last output parameter, not mentioned yet, is an output of a button. A haptic device can provide several buttons similar to the ones of common computer mice. The module can also handle a button and send the information the button was clicked to the framework.

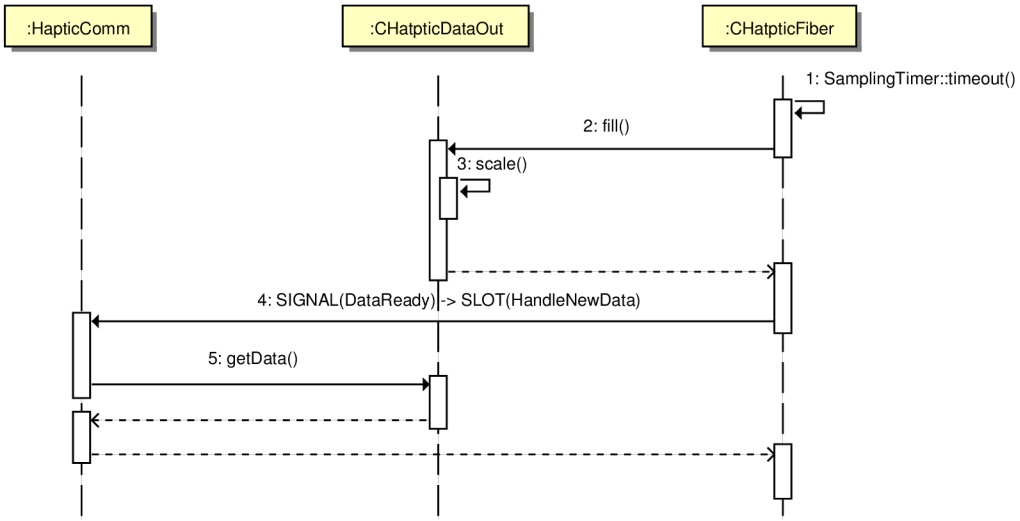


Fig. 3.2: Haptic output data exchange

Figure 3.2 schematically shows the way another software module represented by the class `HapticComm` can get the output data from the haptic module. A software periodic timer determines how often the instance of `CHapticDataOut` is to be filled.

The instance is also responsible for the scaling. After that, `CHapticFiber` emits a signal `DataReaty` to notify the listening module (or modules), that fresh data are available.

3.2 Device Input Processing

The only data the framework may send to the device are values of desired forces in Newtons. The module provides exactly the same possibilities for linear input scaling (including the dead band) as in case of the output processing.

This part provides one more interesting feature - the axes locking. The module can be commanded to lock any of the axis in which the handle moves. If an axis is locked, all the incoming desired forces are discarded and a new force value is computed instead. The new force is generated to restrict the handle to leave the dead band in the locked axis direction.

A simple P controller is used to achieve this behavior. When the distance from the zero of the locked axis is greater than the size of the dead band, the difference of those two values is used as an input for the P controller. The controller's gain is tuned to allow only a very small violation of the dead band.

Even the smallest measurable distance from the boundary of the dead band would result in sending new data, which is not the desired behavior of a locked axis. That is why the module discards also all outgoing data from locked axes.

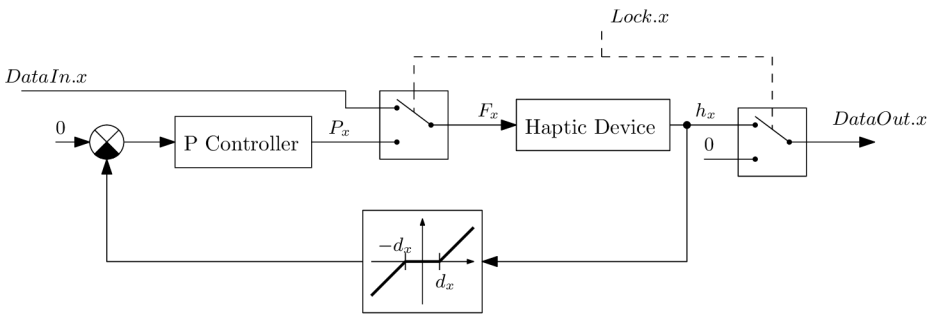


Fig. 3.3: Axis locker block diagram - x axis

Figure 3.3 shows the described locking functionality in a form of a block diagram. The control circuit for the unlocked x axis is displayed. The circuits for the other axes are identical. The haptic device is displayed as one simple block, since its inner control algorithms controlling directly the hardware are embedded into the driver and thus invisible for the module. The symbol $DataIn.x$ represents the force requested by a user, the P_x is the requested force from the P controller, the symbol F_x is the requested force entering the device, the h_x represents the current handle's displacement in the axis, the d_x is the dead band of the axis and the $DataOut.x$ is the output sent to a user. The symbol $Lock.x$ represents the command to lock the axis.

The feature of axis locking is useful, for example, when a user wants to move an actuator only in one axis at the time. He can lock the other axes not to accidentally cause any movement in unwanted direction.

If all the axes are locked at the same time, a side effect is, that the device, in fact, simulates the handle is being inside a virtual cube. The dimensions of the cube are determined by the axes dead band dimensions. The dead band must not be too thin or even of zero size while using the locking, because the too thin dead band may cause oscillations.

The class responsible for the locking is called `CAxisLocker`, it can be seen in Figure 3.1.

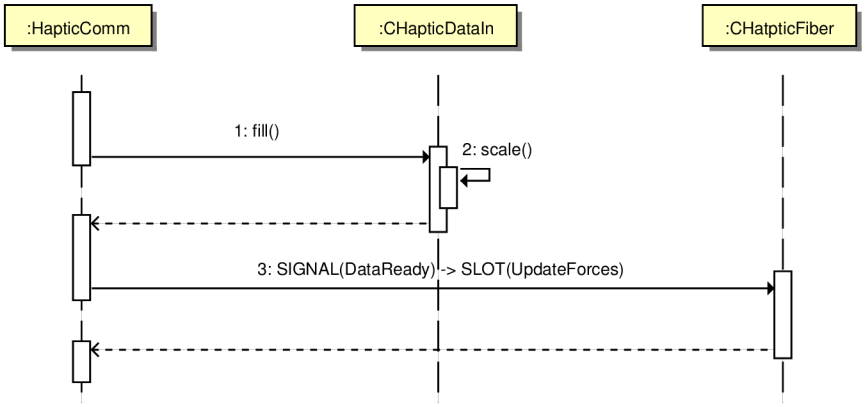


Fig. 3.4: Haptic input data exchange

Figure 3.4 schematically shows the way another software module represented by the class `HapticComm` can send data to the haptic module. After the instance of `CHapticDataIn` is filled, the sending module is supposed to notify the `CHapticFiber` using its slot `UpdateForces`, that new data are available.

3.3 State Machine

To allow the haptic device to run in different modes without any source code changes, this module provides a software finite state machine. Every state represents one mode of operation. Current implementation provides an idle state, a calibration, a normal run and an initialization.

As was already described, the handle's displacement from the zero point is converted to the desired actuator speed. When the program is started, the handle can be in any possible position, so the displacement can be big. Without any safety measure actuators could start moving immediately after the module is started. This behavior is dangerous and highly unwanted. That is why the state machine contains the initialization state. This state moves the handle to the zero point of the device space and holds this position. The movement is based on the same principle as the axis locking - a P controller.

This way of initialization prevents the actuators from unwanted moving after the start, but it introduces a new problem. Without any additional measures, the handle starts moving when the program is started. In an extreme case it could damage itself or an obstacle standing in its way. That is the reason the state machine has the idle state.

The idle state is very simple - no data are grabbed from device (except for the button click) and no forces are generated. So it is suitable to be the starting state. A user is supposed to start the program, hold the handle and switch to the initialization state when he is ready. The idle state may be used in any other situation when the device should be just idle.

The next provided state is responsible for the device calibration. In case of the *Phantom Desktop* device, it means just to move the handle around the device space until it is configured.

It is possible to use this state instead of the idle state at the place before the initialization, because calibration does not generate any forces either.

The last remaining state is the run state. This one represents the normal duty - all measurements are taken and the device forces are generated according to the desired forces.

It is obvious that the particular order of the states as well as the transition conditions must be defined according to current hardware configuration. The source code documentation describes needed modifications of the code in order to adjust the transitions or even the states - see module "User API". An effort was made to make the corresponding code as flexible as possible.

The state diagram 3.5 shows the current state machine configuration. All the state transitions are driven by a single click of the device button.

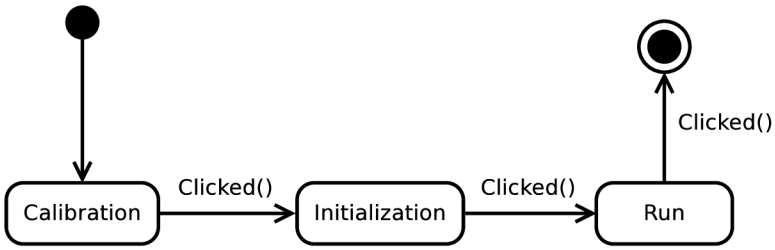


Fig. 3.5: State machine current configuration

4 MACHINE VISION MODULE SOFTWARE DESIGN

The basic requirements concerning the Machine Vision Module were already stated in the Introduction. Let us take a more detailed look now. This module provides a functionality for camera type agnostic image acquisition, for processing of images, for storing them and for visualizing the results. Further in this text the name "image processing chain" is used for all those task together.

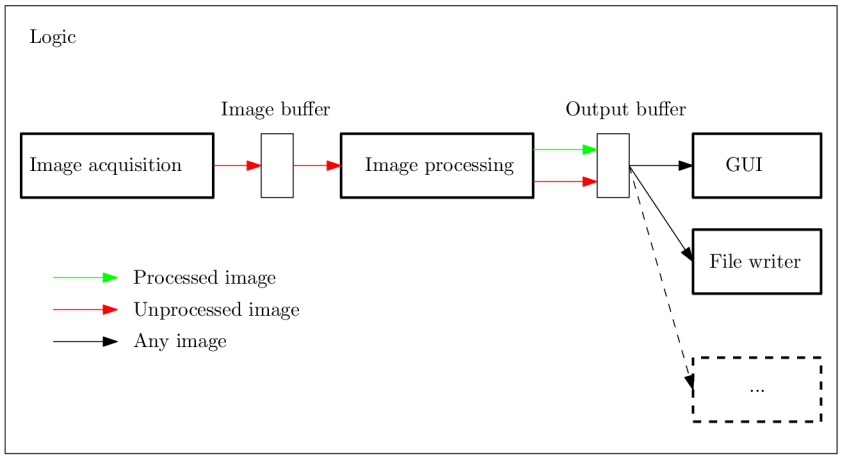


Fig. 4.1: Image processing chain

Figure 4.1 shows basic structure of the image processing chain. Every thick line drawn bar means one thread. The structure is quite self-explanatory except for the fact the main processing thread puts also unprocessed images into the output buffer. A user will most likely want the GUI to show him/her the image from camera and next to it the corresponding image resulting processed image. That is why the second buffer can store two sorts of images.

The next question is, why the processing thread is responsible for feeding the output buffer with unprocessed images. Especially when the GUI (or another consumer thread) could ask for them directly the first buffer and avoid copying data between the two buffers. The problem is, that the consumer threads have lower priority, so they are slow and can be few frames behind the acquisition. In this situation the

synchronization of the ”before” and ”after” images would be very difficult. When the processing itself is responsible for feeding the output buffer with pairs of corresponding images, no complicated image synchronization is needed. The only drawback is the mentioned necessity of copying data from one buffer to the other. The copying overhead is negligible compared to the data processing overhead, Section 5.2.1 gives a proof.

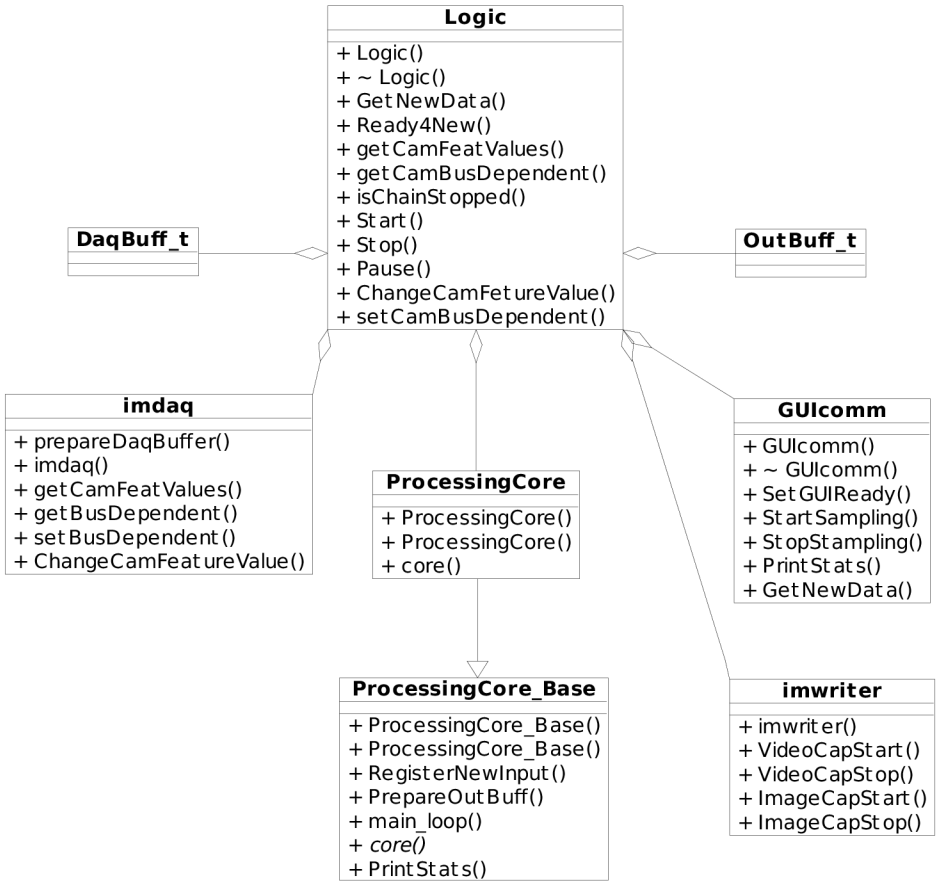


Fig. 4.2: Image processing chain - class diagram

Figure 4.2 shows almost the same as the chain block diagram in Figure 4.1, but expressed by UML class diagram. This diagram is here to connect the provided chain description and the implemented source code. The `imdaq` class represents the image acquisition, the `ProcessingBase` and the `ProcessingCore` represent the image processing, the `GUIcomm` is a part of the GUI and the `imwriter` corresponds

to the file writer. The remaining classes `DaqBuff_t` and `OutBuff_t` represent the image buffer and the output buffer.

This module depends on the OpenCV library, which is an open-source library for image processing briefly described in Section 2.4. The module is supposed to run in the Xenomai environment, but could be ported to another environment by following recommendations in Section 4.7.

Section 4.1 deals with the image acquisition part. Section 4.2 discusses the image processing part design including the current algorithm and future plans. Section 4.3 describes the design of the buffers. Section 4.4 deals with the GUI design. Section 4.5 discusses the design of the image writer part. Section 4.6 describes real time capabilities of the module. Section 4.7 deals with the problem of code migration to a different platform. Finally, Section 4.8 describes the way of the error handling in the module.

4.1 Image Acquisition

The first step of the processing chain is the image acquisition. It is supposed to support as many camera types as possible. The OpenCV provides an abstract layer encapsulating many commonly used cameras, but unfortunately at least in the current version 2.1.0 this layer has several serious drawbacks. The biggest one is that the possibilities of camera parameters configuration are very restricted. For example there is no possibility to set a bus speed for *FireWire* cameras. Although the support for many camera types is required, the framework is currently used mainly together with the FireWire cameras. Speaking about FireWire cameras, there is another problem to be mentioned. A low level function used in the OpenCV to grab data from a camera waits for the data without any timeout. So when the camera stops to send data, e.g. because of some error, the function will block forever, which is not acceptable.

This was just an example, there are a few more similar issues common for all camera types. It is impossible to ask camera for a list of supported video modes etc.

This information implies that OpenCV is not suitable for image grabbing. Now there are two possibilities. The first one is to write our own camera module from scratch and the second one is to modify the OpenCV. The first approach means lots of work and takes lots of time spent writing and especially testing. The second one means to create our own version of the OpenCV possibly incompatible with its newer releases. Fortunately, the OpenCV classes, which need the modification have important methods defined as virtual. So I decided for a compromise - to copy these classes to my project, to inherit from them and finally to redefine those important members. This solution uses existing tested code of the OpenCV and is also compatible with its newer versions, at least as long as they do not change the inner structure dramatically.

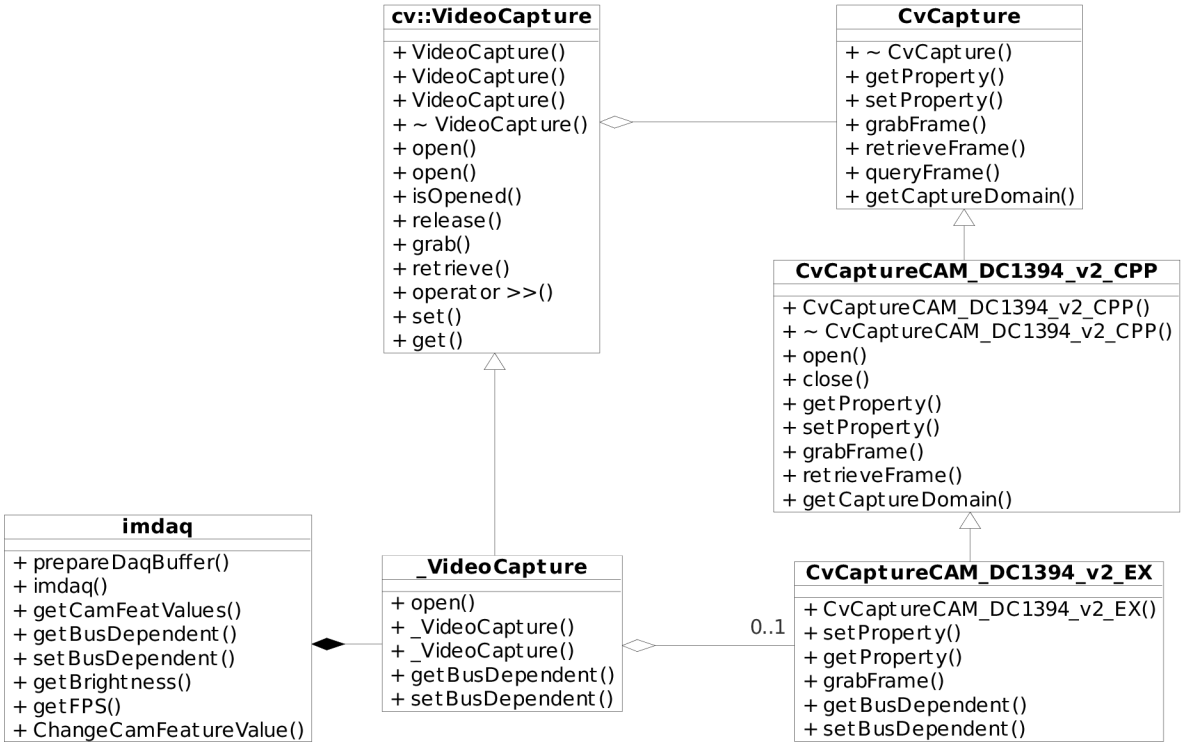


Fig. 4.3: Tailored OpenCV classes

Figure 4.3 shows relationships between the original OpenCV classes `VideoCapture`, `CvCapture`, `CvCaptureCAM_DC1394_v2_CPP` and the newly implemented classes `CvCaptureCAM_DC1394_v2_EX`, `_VideoCapture`. The class `imdaq` is here just

to demonstrate how this part belongs to the chain class diagram 4.2.

The class `VideoCapture` is the mentioned layer providing generalized camera API. More precisely, it provides support also for video files, but that is not important in this project. The class `CvCapture` is in fact core of the `VideoCapture`, because it defines API every particular camera class must implement.

As can be seen the `_VideoCapture` just redefines few methods and adds a new functionality regarding the camera type dependent settings.

As was said already, the most important cameras are the FireWire ones at the moment. That is why all the significant modifications are FireWire related. But that does not mean the module is not ready for the other types, it certainly is. It is legal to use the other ones, but you cannot use the extra settings - e.g. ask the camera for a list of supported video modes, frame rates etc, because those settings are camera type specific and currently were implemented for FireWire cameras only. That is why Figure 4.3 does not show, for example, a *v4l* camera related classes. It shows only the common ones and FireWire ones.

The FireWire modifications are represented by the class `CvCaptureCAM_DC1394_v2_EX`, which redefines few methods from the `CvCaptureCAM_DC1394_v2_CPP` and adds also a few extensions.

Anyway, the code documentation is written to make the developing of the bus dependent settings functionality for custom camera type as easy as possible - see "Tailored OpenCV Parts".

The thread running the data acquisition has together with the image processing one the highest priority. It also should be real time capable, but there is a serious complication, described in Section 4.6.

4.2 Image Processing

The image processing is supposed to be user defined. The main task here is to find a compromise between the amount of the developer's effort and the user's effort. The envelop should provide the user a possibility to define the algorithms as easily

as possible and, on the other hand, cost the developer as little effort as possible. Obviously, to lower the effort on one side means to increase the effort on the other side.

In the end, I decided for the easiest way for a developer and thus the hardest for a user. The main reason was the complexity of the other possibilities and the limited amount of time. At least Section 4.2.2 discusses the other possible solutions.

The chosen solution means to create a base class providing all the functionality needed for the processing but not providing the processing itself - the `Processing_Core_Base` class in Figure 4.2. A user is supposed to inherit from this base class and define one virtual method, which is the core of the processing. It is called every iteration while the supporting actions are done behind the scene and a user need not care about them. This derived class is the `Processing_Core` in Figure 4.2.

So the summary is, that currently the user has to write the algorithm in C++ and the only simplification is that he need not care about data exchange between threads, synchronization etc. Anyway, he has to obey certain rules not to violate real time behavior. The rules are discussed in Section 4.6.

It is recommended to use the OpenCV framework to implement the custom algorithm, since the chain uses its classes to represent the image data.

4.2.1 Current Algorithm

The current configuration of this module contains a simple algorithm for tracking liquid in a special cartridge. This algorithm was developed as a part of completely different project and was written by someone else. The tracking itself is not important for this project, it serves only as an example how to use this module together with a custom algorithm. The source code documentation says more about this problem and gives hints how to define a new algorithm.

Moreover, this tracking algorithm violates real time behavior of the module, because it contains OpenCV function `cvCalcMotionGradient()`. This function dynamically allocates some of its local objects and it is not allowed to do that in real

time context. Section 4.6 explains how to avoid such problems.

4.2.2 Other Ways to Implement User Defined Algorithm

A more difficult solution for a developer and at the same time more convenient for a user would be to use dynamic libraries. The user defined algorithm would live in a dynamic library and the module could load it. This solution is better than the existing one, because it enables algorithm switching on the fly - the module can unload current library, load another one etc.

Usually, people educated in field of image processing are more familiar with Matlab environment than with C++. A translator from Matlab language to C++ code could be added to the library oriented solution to make it even more user convenient.

Matlab provides so called *Real-Time Workshop*, which is capable of generating C/C++ code from Simulink models and Matlab scripts [31].

Since Xenomai provides also RTAI API via the skins, it would be theoretically possible to use the RTAI-Lab or at least its parts as the code generator.

An alternative to those conversions of Matlab source is to create a set of basic parametrized image processing operations (threshold etc.) and allow a user to build the algorithm from those basic blocks. Flexibility of this solution is obviously low, unless a developer is willing to prepare a huge set of those basic blocks. But even if a developer was willing, to prepare such a set would be very infective in this case.

However, all the described possibilities are supposed to be just a hint for the future developers. The final decision is out of scope of this thesis.

4.3 Buffers and Data Exchange

One of the most important and also one of the most challenging problems is the data exchange among the threads of the processing chain, as illustrated in Figure

4.1. As can be seen from this picture, all needed data exchange is provided by two buffers – the Image Acquisition Buffer and the Output Buffer.

Although both of them hold different data types, their inner structure is the same. They are internally represented as ring buffers (also called circular buffers) based on a linear linked list. Such a structure is very convenient to use, because there is no need to care about index overflow and even about indexes at all. The both buffers are single writer, which makes situation a bit easier. The number of readers is not limited.

Since there is no similar structure provided by Qt or OpenCV, it was whole written from a scratch.

The buffer structure keeps track of a writer pointer (referred as *writer* in the following) and about all reader pointers (referred as *readers*). Before the detailed description how the structure handles those pointers, let us assume that the buffer is meant to be accessed from different threads.

4.3.1 Writer Thread

If a thread calls buffer's writing method to put some data to a buffer, the buffer structure checks the buffer cell currently pointed by the writer. If this cell is pointed by any reader, it is considered as occupied and the writer moves to the next cell. Besides that the writer marks the occupied cell as outdated to prevent the other readers from reading it in the future. Readers skip outdated cells. This is important because of data continuity. The cells before the outdated one contain newer data. So the outdated one is a hole that must be skipped by every reader not to mix old and new data. It is assumed that algorithms using data from a buffer can accept a data loss caused by too quick writer, but they cannot accept a stream of new continuous data corrupt by one old image. Since the buffer is circular, the writer returns to the skipped cell soon and deletes the outdated mark when finishes its filling with new data.

The writer skips occupied cells until it finds a free cell. Then the writer copies

new data to this cell and moves to the next one and the buffer's writing method exits. The advantage of this approach is that the writer does not block the last filled cell, because it shifts to the next one immediately.

The last important remark is, that the buffer is not allowed to put to sleep the thread calling buffer's method working with the writer. Or more precisely - theoretically the writing thread can sleep, because there is a short critical section protected by a mutex, so the thread may, in fact, sleep when it cannot acquire the mutex. But there is nothing like a possibility to put the thread to sleep on a condition variable etc.

4.3.2 Reader Threads

If a thread calls buffer's reading method, the method identifies the right reader and then checks the cell after the one pointed by the reader. If it is not pointed by the writer, the reader is shifted to it. Otherwise, the reading thread is put to sleep and woken up by the writing method as soon as any new data are written.

If the reading thread is not put to sleep, the method checks the outdated mark. If it is set, the method skips the cell by repeating the so far described procedure. Otherwise the reading method returns a pointer to the cell and thus exits.

So the reader method does not copy any data, it just returns a pointer to them and protects the cell against rewriting until the next call of the reading method occurs.

The reader method never cares about other readers, because one cell can be read by multiple readers at the same time without any restrictions.

4.3.3 Summary

This reader writer oriented approach has one big advantage, because the thread synchronization on the buffer means just to protect by a mutex only the short parts comparing the writer and reader pointers. In another words - there is no need to protect reading or writing.

After reading the two previous sections, a question may arise - why the writer blocks readers when it is one cell ahead, instead of when it is at the same cell? That is because of a smoother initialization. The buffer must be initialized to state when the writer blocks the readers since there are no data at the very beginning. Let us imagine the situation when the writer blocks readers pointing the same cell. So after the initialization all the pointers point the same cell. When the writer tries to fill this cell it has to skip it and mark it as outdated. The readers skip outdated cells, so they will skip the very first cell, as soon as they start to work. The "writer ahead" method has a smoother start, because there is no useless skipping.

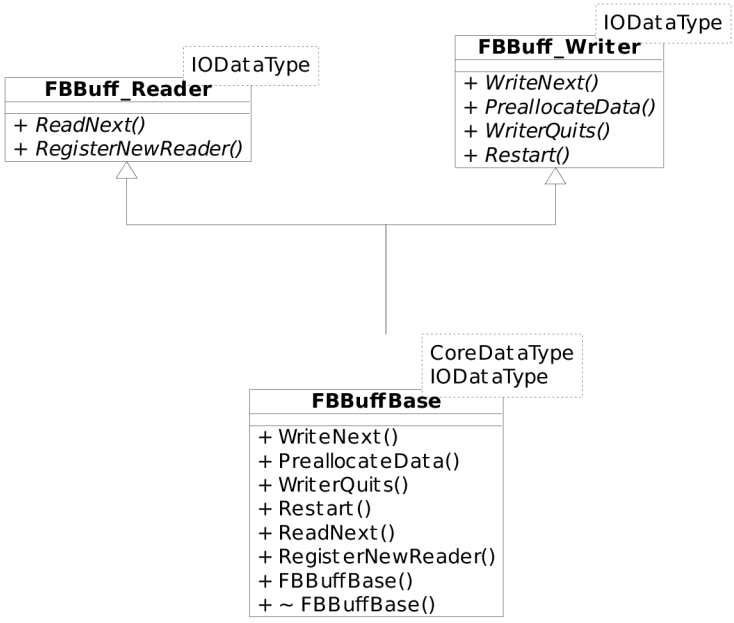


Fig. 4.4: Reader writer access model - class diagram

To provide a more comfortable user interface and to make a work with this kind of buffers a bit safer, the implementation provides two data types to represent the reader and the writer functionality separately. Figure 4.4 shows the basic idea. It is possible to use a pointer of a base class type to access an instance of an inherited class. So a thread on the writing side owns a pointer of type `FBBuff_Reader` and threads on the reading side own a pointer of type `FBBuff_Writer`. This safety measure ensures that a writer thread cannot call the reading methods and a reading

thread cannot call the writing methods. To call the writing methods from a reading thread (or vice versa) may cause a deadlock, because of a thread waiting for itself. This approach also prevents a reader threads from modifying a buffer and helps to ensure that only one writer side exists.

4.4 GUI

This module also contains a standalone graphic user interface. The GUI is standalone, because the module has not been completely embedded into the CoSMic framework yet. The GUI's purpose is to demonstrate the current capabilities of the module.

The GUI provides basic settings widgets and a control functionality - the whole processing chain can be started or stopped by a single button. It also visualizes the grabbed and the processed images.

As discussed in Section 4.1, there are two sets of settings. Settings supported by OpenCV and then camera type dependent ones. That is why the GUI provides two settings widgets. Again, the camera type dependent widget was implemented only for the FireWire cameras. If this module is run with any other camera type, this widget will automatically hide itself. There are hints on how to implement similar widget for different cameras, in the source documentation (see `EXsettingsFW` class reference).

Anyway, the most beneficial part of the GUI is the widget to display images - to display the OpenCV native image format in a Qt based GUI. Since both of those frameworks are quite popular, this widget can find its place also in another future project.

So far it was mentioned just as "widget" but, in fact, there are two widgets. Their functionality is the same, but one is based on software rendering (class `OCVdisplay`) and the other one on Open Graphics Library (OpenGL) rendering (class `OCVdisplayGL`). The advantage of the OpenGL version is that it moves related load from

a CPU to a Graphical Processing Unit (GPU), but unfortunately there is a serious drawback. This module is supposed to run on Linux systems and it is a well known fact that quality of certain Linux GPU drivers is poor [32], and so can be the OpenGL support. The result is, that the OpenGL widget may work slowly or even refuse to work on certain system. The version without OpenGL should run everywhere.

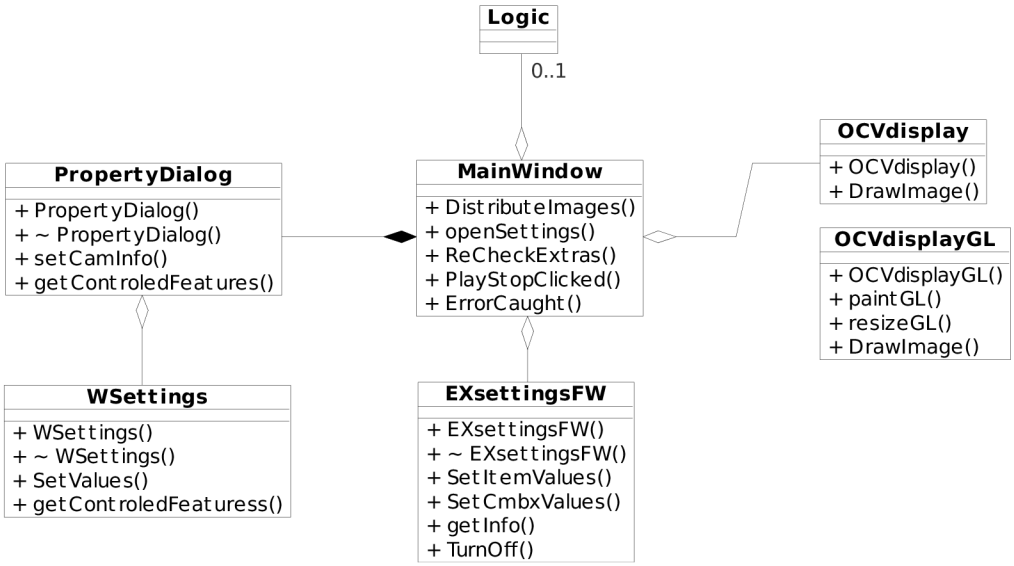


Fig. 4.5: GUI class diagram

Diagram 4.5 shows the main GUI classes. The class `Logic` is here just to demonstrate the relationship between the GUI and the processing chain. The discussed widgets can be identified in the diagram by the class names quite easily - the only confusing name could be `WSettings`, which is a widget handling the basic set of settings.

The diagram is a little bit inaccurate for the sake of simplicity. All the aggregations (except for the one between `Logic` and `MainWindow`) are not that direct. Since the used IDE (Qt Creator) provides a GUI design support that can generate pieces of source code automatically, there is one more layer in the middle of the aggregations.

4.5 Image Writer

The last part of the image processing chain is the Image Writer. Its task is obvious - to store images (no matter if the processed ones or the ones directly from camera) in form of file sequences or a video file. The thread the writer is running in has together with the GUI thread the lowest priority.

OpenCV provides classes for convenient writing of OpenCV images to a harddrive in one of the typical formats. So the code of this part is very short and simple.

4.6 Real time Capabilities

As mentioned in the introduction, this module was designed to fulfill certain real time requirements. Although it is based on the Xenomai Linux extension (Section 2.1.2) it cannot be fully real time. The reason is that, there is no suitable FireWire driver compatible with the current version of Xenomai (or Rtai). This situation forces the module to use the ordinary Linux driver, which is not designed to be real time. To call the driver related kernel services means to switch the execution to the Xenomai secondary mode (Section 2.1.2).

The maximal care was taken not to create any new problems preventing the module from running in real time. If there is a suitable driver in the future, the module will run in real time. Obviously, certain modifications of the image data acquisition part will be needed to make it compatible with the new driver's API, but this is not important right now.

Important is to keep in mind, how to modify the existing code or how to define a custom image processing algorithm in a real time friendly way. The rule number one says, that it is prohibited to call any native Linux kernel service. Those services are usually called via library functions like `printf()`, `pthread_create()` etc. Use Xenomai equivalents where possible or avoid those functions completely.

The rule number two says, that one must avoid dynamic memory allocation. There are lots of data to allocate in this module, but all the allocations are done

before the main loops of the threads. All is pre-allocated.

The first advice to avoid dynamic allocations is to pre-allocate everything possible. Unfortunately, this advice is useless for the user image processing algorithm, because there is no suitable place for the pre-allocation in this part. The second possibility is to use the allocation routines provided by Xenomai, explained in the related part of the API documentation [33].

Anyway, the two approaches are suitable only for a custom code. The situation is even more complicated when a user wants to exploit a third party library, for example OpenCV. Then there are only two options. First, to modify the code of a library to force it to use pre-allocations, or the Xenomai heap management. The conversion to the real time heap management can be done by overloading the `new` and the `delete` operators.

The second option on how to use a third party library is just to avoid all functions that allocate memory dynamically.

Fortunately, Xenomai (and thus this module) provides functionality to detect real time behavior violating spots - see `XenoSwitchWatchdog` class reference.

4.7 Code Migration

Every operating system provides its own API for thread management. This module provides an abstract layers unifying certain parts of the threading related API. That significantly simplifies a migration of the whole module to a different operating system. The only action that has to be taken is to write a short code connecting the selected system's threading API with the abstract layers.

Currently, the module can be switched between Xenomai and Linux mode. The popular tools like *Valgring* are not Xenomai compatible. That is why the pure Linux port was made.

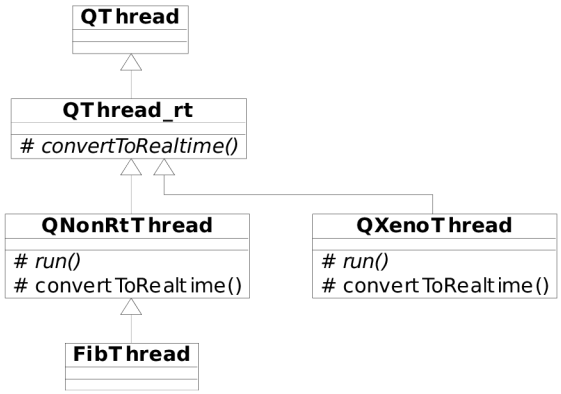


Fig. 4.6: Unified thread representation - class diagram

4.7.1 Unified Threading

As can be seen in Figure 4.6, the whole layer is based on the `QThread` class. This fact itself provides a certain level of generalization, because Qt can run at Linux, Windows and Mac OS X. So the portability among the major operating systems is achieved, but we need to add something more to cover also real time systems. Paradoxically, here the `QThread` happens to be a limiting factor, because it restricts possible RT systems only to RT extensions of the mentioned major operating systems. But that is acceptable in this project, since we consider only the RT extensions of Linux.

The next class introduced in Figure 4.6 is `QThread_rt`. It just adds one method - the method to convert an ordinary thread to RT thread. This level of the diagram defines the API.

The most important inheritance level in Figure 4.6 is the one providing classes `QNonRtThread` and `QXenoThread`. Those classes are two implementations of the API defined by the previous described classes. If someone wants to add a support, for example, for the RTAI API, he needs to add his own class to this level.

The last level contains the class meant to be instantiated in the program. This class is in fact the unified layer. In the situation described in Figure 4.6 it inherits from `QNonRtThread`, but the trick is, that this inheritance is conditional, based on preprocessor macros.

This model has a huge advantage. It allows to extend the set of supported APIs very easily and at the cost of minimal changes of the existing code.

No matter of the chosen implementation, the usage of the class `FibThread` is, in fact, the same as the usage of the original `QThread` - the only difference is that `FibThread` provides the function allowing to convert a running thread into a real time thread.

The source code documentation provides more details about the abstract layer - see "Threading support".

4.7.2 Unified Synchronization

The module provides the class `FibThread` representing a generalized thread. The project needs one more feature - unified synchronization objects. Currently, only two types are in use - mutex and condition variable. Their generalization is done in the same way, so let us focus on the mutex case only. If you compare Figure 4.6 and Figure 4.7, it is obvious, that the latter one is missing the lowest layer - the equivalent of the `FibThread` class. That is because the design of the generalized mutex is older that the thread related one. This older idea is less comfortable but the effect is the same. There is the class defining API and the classes providing the implementation. Since there is no class encapsulating the API selection via the conditional inheritance, we need pointers to the base class `FBMutex` and to allocate an instance representing the chosen API. The allocation is conditional using preprocessor directives.

The drawback of this approach is, that if someone wants to add a support for a new API, he needs to change more places in the code.

4.8 Error Handling

Usually, there are two ways of program errors handling - the C style, when a function return value determines whether the function was successful, and the C++ style

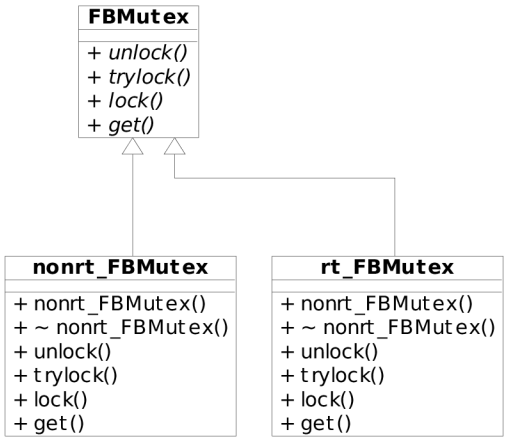


Fig. 4.7: Mutex representation - class diagram

using exceptions. Qt provides the signal technique (Section 2.3.1), so it is also possible to emit a signal in case of error. That is quite similar to the exceptions, but the signals have one huge weak point - constructors. The only way one can handle errors in constructors is to throw an exception. It is impossible to emit a signal from a constructor, because the related connection was not established yet. One cannot connect signals/slots between objects that are not constructed yet. So the connection has to be established after the call of a constructor and that is why one cannot emit signals from constructors.

The superior CoSMic framework uses signals for the error handling. So this module has to use them too to make the integration easier. I decided to use exceptions internally thanks to the constructor issue. There are places in the code catching exceptions thrown from lower layers and converting them into signals.

Unfortunately, this error handling compromise is not capable of converting all possibly thrown exceptions. The current main constructor (`MainWindow` class) may throw an exception. But this is the only place - the rest of the module does not throw anything.

Previous Chapter 3 does not include a section similar to this one, because the haptic module does not use exceptions at all. It uses signals for error handling.

5 EXPERIMENTS

Several tests were taken to verify capabilities of the implemented modules. Since the task of the haptic module is rather simple, only one test concerning haptic device was taken. The rest of the tests is aimed to the vision module.

All the tests were run on the same computer. Its brief hardware configuration is summarized in Table 5.1. For further information, see the enclosed file containing the output of the *lshw* command. In this configuration, OpenGL works correctly

Tab. 5.1: Testing computer's hardware configuration

CPU	Intel Core 2 Duo CPU E6750
Width	64 bit
Cores	2
Memory	3 GiB
GPU	GeForce 8400 GS
Chipset	Intel Q35

only with the proprietary GPU driver from the manufacturer. Unfortunately, there was no time left to make this driver run together with the Xenomai kernel. All the test were performed using the software rendering.

5.1 Haptic Experiment

The purpose of this test was to verify all the implemented functionality the module provides. The equipment consisted of the *Sensable Phantom Desktop* haptic device, three actuators, a stabilized voltage supply and the testing PC. The haptic device controlled the actuators and received values of desired forces from another software module responsible for a voltage measurement. The desired forces were proportional to the voltage level.

An operator was manipulating the handle and watching whether all the actuators were moving correctly. Then he was changing the level of the analog voltage and checking whether the device was generating corresponding forces. During both of the tasks the operator was also locking and unlocking the axes randomly.

The test was successful and proved all the implemented functionality including axis locking worked, but it revealed an issue, that deserves to be noted here.

The haptic module is relatively CPU time demanding because of the software control loop embedded in the device driver and in the 3dTouch library - i.e. not created in this project. The running module took circa one core of the processor. When a computer is serving the device under a heavy load (other than caused by the control loop itself) the inner closed loop cannot fulfill certain deadlines and the device starts to operate in an unpredictable way. Moreover, the library is not robust enough and there is a high probability of crash due to segmentation fault under these conditions.

One could argue that an RT operating system should be used to solve this issue, but the 3DTouch library comes only in two versions - for Windows and for Linux. None of them is an RT operating system. Even if one decides to deploy any of the described RT extensions, the library cannot run in the RT mode without appropriate source code changes.

The resume is, keep this in mind and do not run any other CPU demanding tasks on the computer dedicated to the haptic device.

5.2 Machine Vision Experiments

This section groups all experiments performed to verify Machine Vision Module's capabilities and to learn more about its performance.

The first experiment is described in Section 5.2.1 and its purpose is to find CPU time expensive spots in the code using the method of code profiling. The second experiment described in Section 5.2.2 verifies whether the memory management does not restrict the RT behavior of the module. The next experiment explained in

Section 5.2.3 compares the times needed to perform certain task of the module in Xenomai and in Linux. Finally, the last experiment discussed in Section 5.2.4 deals with a possible image loss among the processing chain parts.

5.2.1 Code Profiling

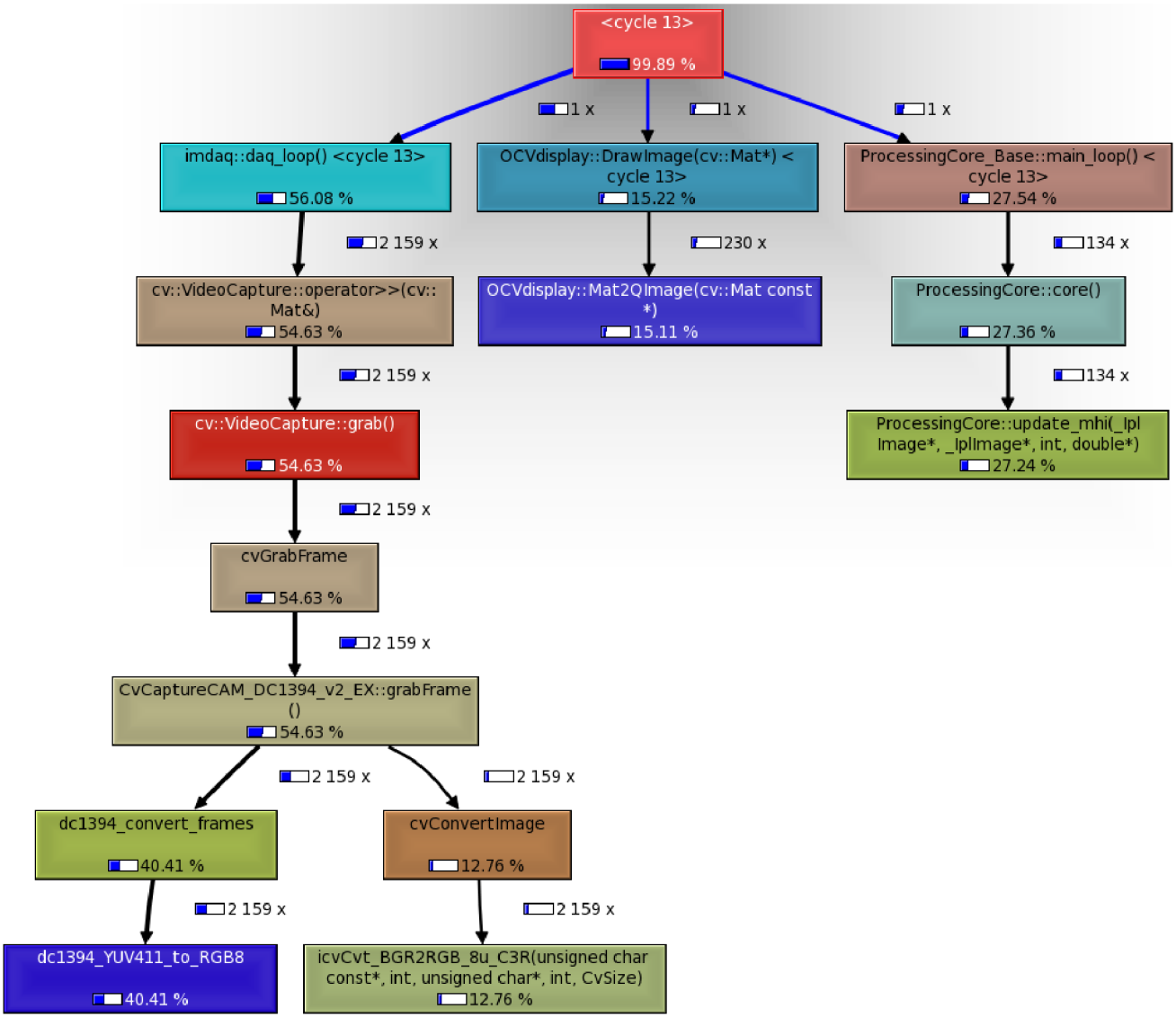


Fig. 5.1: Simplified call graph - *Sony DFW-V300* attached

Code profiling means to determine how long a program spends in certain routines and how often calls them [34]. In this experiment the tool called *Callgrind* [35] was

used to profile the code and the tool *KCachegrind* [36] to visualize the results. Callgrind is a member of the *Valgrind* family, which is a framework grouping several tools for a dynamic analysis of a code [37].

This test is based on the code profiling of the Machine Vision Module and it aims to find whether there is a place for any code optimizations resulting in a better module performance. The module was run for certain time with a color camera *Sony DFW-V300*. The camera was not focused on anything special, since the current image processing algorithm is in the module only as a placeholder, so its results were not important.

Figure 5.1 shows resulting call graph. Every block represents one function, except for the block *<cycle 13>*, which is used by *KCachegrind* to group cyclically called functions. The number inside of each block denotes how many time the program spent inside the function. Arrows mean function calls and the numbers next to the arrows say how many times the call was performed. This call graph is simplified. Only functions that cost more than 10 % of the overall program run time are displayed.

Figure 5.1 tells that the most CPU time expensive part is the image acquisition (`daq_loop()`). It is even more expensive than the whole image processing (`main_loop()`) and than the GUI (`DrawImage()`). The main purpose of a profiling is to identify places in a code that have a negative impact on a performance (so called *bottle necks*). After another look at the graph, such a bottle neck in the image acquisition can be found - `dc1394_YUV411_to_RGB8()`. This function is responsible for the image data format conversion. Different cameras may send images in different formats, so a conversion to one common format is needed and this format has to be also compatible with OpenCV. The used camera *Sony DFW-V300* sends images in the *YUV411* format, so the function converts them into the *RGB8* format [1]. As can be seen, this conversion is the most expensive operation in the program.

However, this conversion is necessary and there is not much one can do with the

[1] To be precise - OpenCV uses *BGR* format, but the next conversion from *RGB* to *BGR* is very cheap compared to the discussed one.

function. It is a part of an open-source library, so theoretically there is a possibility to modify it and optimize it, but that would cost lots of effort and a probability that the library is already optimized as possible is high.

There is a very simple solution to this performance issue. One cannot change the function, but one can change the camera. This is a good place to remember the module is camera type agnostic, so the camera change will result in a change of the conversion procedure automatically. The next call graph in Figure 5.2 describes the same situation as the previous one and also obeys the "above 10 %" rule. The only difference is the camera used. Now it is *Sony XCD-X710*. The resulting call graph

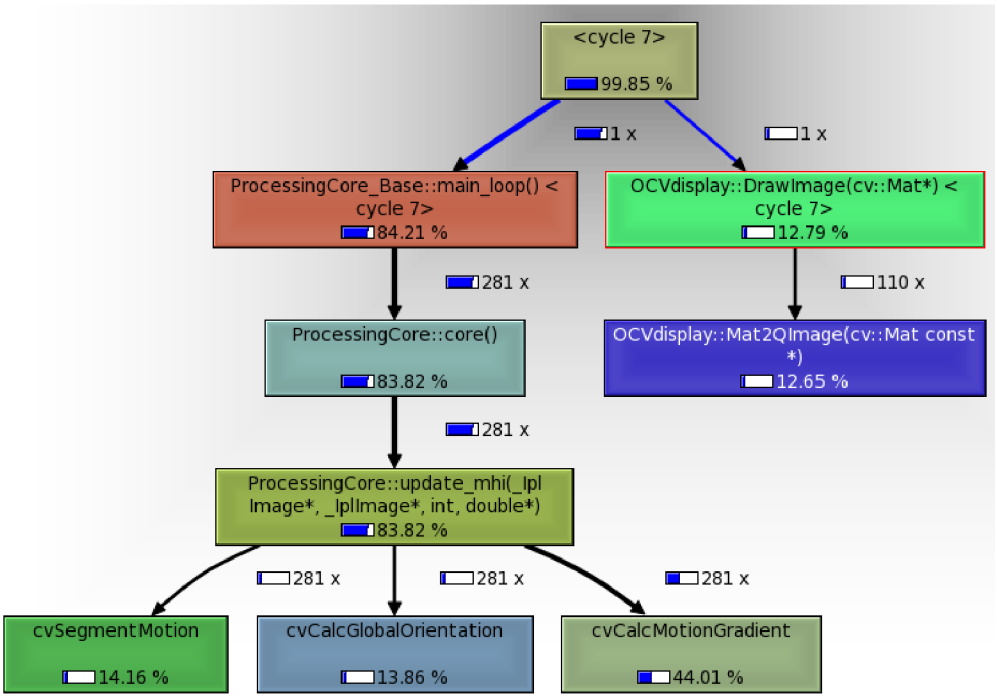


Fig. 5.2: Simplified call graph - *Sony XCD-X710* attached

in Figure 5.2 is much simpler and especially does not contain the image acquisition branch at all. It was pruned away because it takes less than 10 % now. The used camera is monochromatic and thus can be asked to send data in the 8 bits per pixel mode. To convert this format to the one OpenCV uses to represent monochromatic images is very simple and especially cheap.

This shows how important is to pick right camera, set up in right mode of operation.

Remark on Data Copying

Both of the call graphs show also one more fact. Section 4 mentions the theoretical problem of expensive data copying between buffers. The function responsible for all significant copying in the module is `memcpy()`. As can be seen, none of the graphs contains this function. That means the program spent less than 10 % of its execution in the function. The enclosed full outputs of the profiling tool prove that the share of the function was around 3 %. So all the copying is really negligible compared to the image processing and to the rest of the module.

5.2.2 Pre-allocation Verification

Another useful member of the Valgrind family is the heap-profiler *Massif* [38]. It shows how an analyzed program manages its memory during its run. As discussed in Section 4.6, the right dynamic memory allocation is very important in an RT environment. That is why the Machine Vision Module pre-allocates all image data before the point the threads are switched to the RT mode. The aim of this test is to prove the pre-allocations work correctly.

Figure 5.3 shows a piece of Massif's profiling output, it is memory usage versus time[2] diagram. The meaning of the characters Massif uses to build the diagram is not important here. The only aspect that matters is the shape of the diagram.

The diagram was created under the following conditions. The program was started in time 0, then the image processing chain was started (time `A`), in time `B` the desired size of the taken images was lowered and finally the program was killed in time denoted as `C`. If a user changes the desired image size, the chain has to be stopped first. Then the module performs a new pre-allocation before it starts the chain again. So there should be two rapid changes of the allocated memory amount

[2]The time is expressed in number of executed instructions.

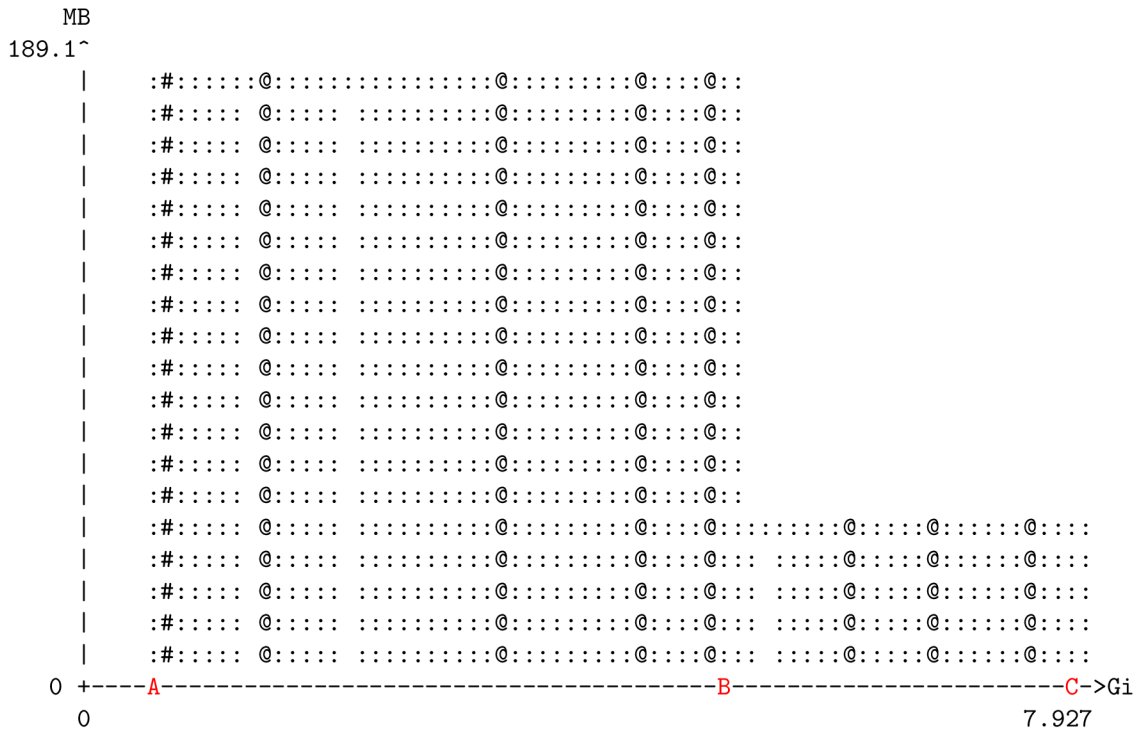


Fig. 5.3: Memory allocation profile - pre-allocation enabled

around the points `A` and `B`, because of the related pre-allocations. That is exactly what can be seen in Figure 5.3.

But this is not yet the promised proof. The proof is the obvious difference between discussed Figure 5.3 and Figure 5.4. Both of them show the similar situation, but the latter was created with disabled pre-allocations.

Disabled pre-allocation means that the data are allocated when they are really needed. The memory is allocated step by step in smaller pieces for a certain amount of time. Figure 5.4 demonstrates this behavior. The curve of the diagram starts to increase slowly in the spot `A` and to decrease again in `B`.

To conclude this discussion - a working pre-allocation means a rapid change of the used memory in a short time. The situation without a pre-allocation is characterized by many smaller changes spread around a longer time interval, because the memory is allocated when it is needed. Figure 5.3 and especially the attached Massif's output file, which was the diagram taken from, prove the pre-allocation approach

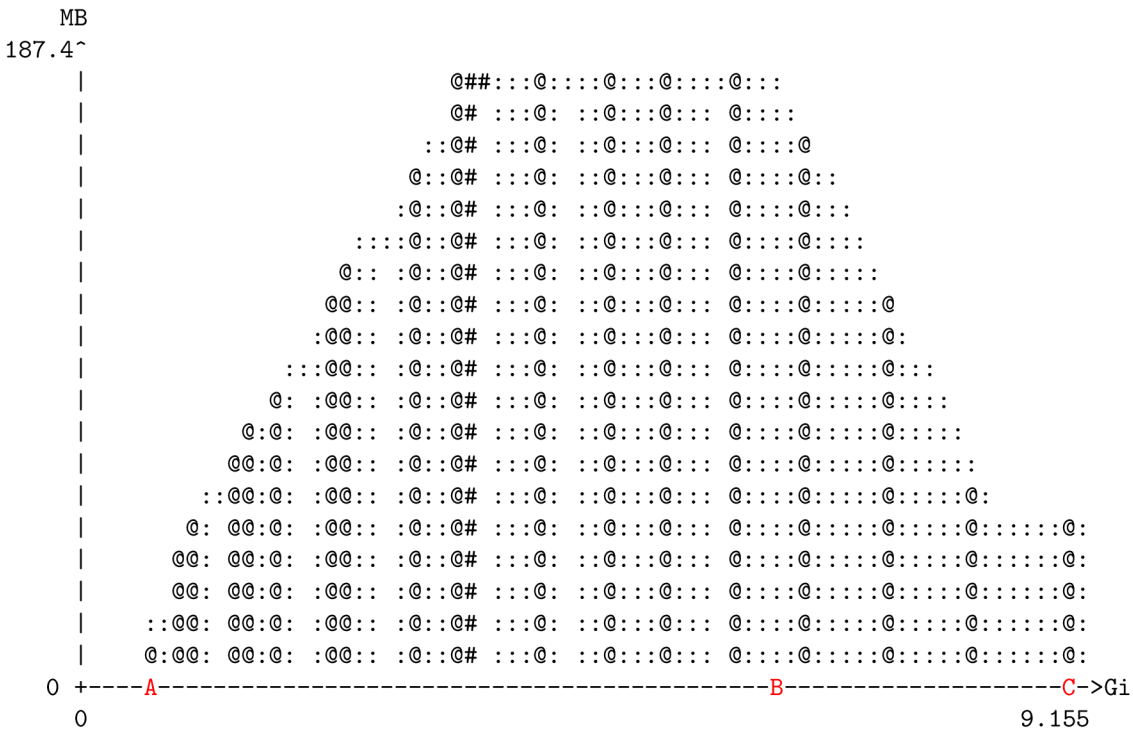


Fig. 5.4: Memory allocation profile - pre-allocation disabled

works correctly.

One last remark - the corresponding spots `A`, `B` and `C` are not exactly at the same places at the time axis of the figures. That is because the program was controlled by a human during the test, so it was impossible to achieve a precious timing. Anyway, a precious timing is not needed here.

5.2.3 Processing Times

The aim of this experiment is to compare times needed by the module to perform certain tasks, while running in Xenomai and in Linux environment. This measurement is not meant to be a scientific proof of the module qualities in the Xenomai mode. Its purpose is to find out, whether one can expect a better time related behavior of the Xenomai version even despite the restriction of the RT capabilities described in Section 4.6.

This experiment is based on a measurement and a statistical evaluation of two

time intervals in the running module. The first interval T_1 represents the time between the finished image acquisition and the finished image processing (the point when the processed image is in the output buffer already). The interval T_2 determines the time between the finished image processing and the finished drawing by GUI. The statistical evaluation means, that the module also computes the mean value and the sample standard deviation of the measured intervals.

No special tool was used, since the module provides the desired information. Two sets of measurements were taken. The first was performed with the module in the non RT mode and the second one with the module in the Xenomai mode. In both cases the module was let running for about twelve hours with the color camera *Sony DFW-V300*, configured to send images with resolution 800x600 and framerate 7.5 fps. The camera was not pointed at anything special. Table 5.2 shows the results of the first set and Table 5.3 the results of the second one.

Tab. 5.2: Statistical time measurement based on 418633 samples, Linux mode

	t_{min} [ms]	t_{avg} [ms]	t_{max} [ms]	s [ms]
T_1	18.1844	34.0191	219.2550	$2.1922 \cdot 10^4$
T_2	36.0622	58.6217	337.573	$3.4305 \cdot 10^4$

Tab. 5.3: Statistical time measurement based on 436627 samples, Xenomai mode

	t_{min} [ms]	t_{avg} [ms]	t_{max} [ms]	s [ms]
T_1	28.0656	48.6059	98.2234	$1.1183 \cdot 10^3$
T_2	44.5772	68.2044	209.7330	$1.2038 \cdot 10^3$

After a comparison of those two tables, it is obvious that the ranges $< t_{min}, t_{max} >$ are shorter and the standard deviations are smaller in case of the Xenomai results. Now a question can arise - how is it possible to achieve such a huge standard deviation in the RT environment? First, there are certain restrictions of the RT capabilities described in 4.6, but even if there were no such restrictions, the standard

deviation could be high. The reason is that the image processing algorithm does not work in a constant time. The time of its executions depends on the quality of input images - not on operating system's RT capabilities.

However, the interval T_2 is independent on the time of processing and its standard deviation is huge too. That is true, but the GUI thread runs with the lowest priority, in fact not in RT mode.

There are still two more possible questions. First, why are the Xenomai results slightly better (shortest ranges $< t_{min}, t_{max} >$ and smaller standard deviations) despite the restrictions mentioned in Section 4.6? The main restriction of the RT behavior is the absence of the RT driver causing switches to the Xenomai secondary mode. But that does not mean the thread exits the primary mode for good, it can be entered again by calling any Xenomai service. A program exiting the primary mode periodically cannot guarantee bounded maximal processing times, but as the tables show, it is still able to achieve shorter maximal values than a program running completely in the secondary mode.

One last question remains. Why are the mean values of the time intervals smaller in Linux than in Xenomai? It is important to realize, that "real time" need not mean "quick" [5]. Since a RT system is designed to fulfill certain deadlines, its inner structure logically has to be more complicated than the structure of a non RT system. That is why a performance of a RT system may be worse than a performance of a comparable non real time system. All the changes making the system real time cost some CPU time. The discussed issue of the higher means in Xenomai environment is a good example of this behavior.

A weakness of this test is a fact, that there was no defined scene to be a reference input for the camera. The input was random, but for a significant amount of time it was static, because the measurements were done partially during the night. Anyway, there is no guarantee the input was the same during the both measurements and different inputs can result in different times the image processing takes. On the other hand, the interval T_2 is independent on the time of processing, as noted already. So

the T_2 should be more reliable.

The reason why a reference image was not used as the common input for both of the experiments, is that this would mean to bypass the image acquisition. Since the image acquisition part contains the RT related issue described in Section 4.6, it had to be included in this experiment.

This test proves, it makes sense to use Xenomai for this module even despite the restriction of the RT capabilities described in Section 4.6. More precisely, it makes sense when a bounded interval of possible processing times is needed. When one prefers shorter average processing times, the Linux solution is better.

5.2.4 Processing Chain Data Loss

If a writer is putting data to a buffer quicker than a reader is consuming them, a data loss may occur. The image processing chain contains two buffers, so there are two possible spots, where a data loss may occur. The purpose of this experiment is to determine how serious the losses are.

The module uses image indexes to determine how many images were lost. Table 5.4 summarizes data losses between the image processing part and the GUI. Data losses between the data acquisition and the processing are not displayed, because there were not any during the test. The test conditions were very simple - running module using the *Sony DFW-V300* camera. All possible combination of frame rates and image resolution were tested. The "-" sign in the table denotes a combination which is not possible for this camera.

The GUI thread has a low priority and it is acceptable for it to lose a certain amount of images, but the numbers in the last row of Table 5.4 are unacceptable - the GUI cannot afford to lose over 40% of incoming images. But there is one additional fact, that should be mentioned. The GUI takes data from the buffer with constant period, which was at the time of the test set to 100 ms. Logically, if there are 30 incoming images per second and the drawing rate is 10 per second, a data loss is imminent.

Tab. 5.4: Data loss dependency on frame rate and image resolution

fps [s^{-1}]	Image loss [s^{-1}]		
	160x120	320x240	640x480
3.75	-	0	-
7.5	0	0	0
15	4.8	4.8	8
30	12.8	12.8	-

The same test was taken with the module compiled in Xenomai mode, but the results remained the same.

It would be interesting to repeat this test with the GUI sampling set to some reasonable value. Another interesting test would be to observe the dependency of the values in the table on the priority of the GUI thread.

This experiment reveals how important the mentioned sampling period is. Although it is probably not evident, the most important result of this test is the proof there was no data loss between the data acquisition and data processing parts. This result is very positive, because any data loss between those two parts is unacceptable.

5.2.5 Summary

The first experiment in Section 5.2.1 is dedicated to the code profiling of the module. The experiment shows that the conversion between a camera native image format and the format used in the module can be surprisingly CPU time expensive. It also shows that the CPU load caused by the data copying performed in the module is negligible compared to the data processing load. So there is no need to avoid the possibly problematic copying discussed in Section 4.

The experiment from Section 5.2.2 verifies the functionality of the memory pre-allocation measures. It discusses a memory profiler output to successfully prove the measures work correctly.

The experiment discussed in Section 5.2.3, deals with the image processing times. It showed that in the current configuration Xenomai was able to ensure shorter maximal processing times even despite the RT restrictions explained in Section 4.6. On the other hand, the experiment shows that a non RT solution should be chosen instead, if the shorter average processing times are more important.

The Data Loss experiment described in Section 5.2.4 showed that there were no image losses between the image acquisition and the image processing parts. Also, it revealed the important fact, that it was a bad idea not to adjust image sampling timer of the GUI to a current frame rate.

6 CONCLUSION

This chapter summarizes all the issues discussed so far and also proposes steps to take in order to improve the existing implementations.

6.1 Achievements

The most significant contribution of the thesis is the design and the implementation of the two software modules extending the functionality of the CoSMic framework - the Haptic Module and the Machine Vision Module (also called ViCo). The both modules were written in C++ language using the Qt framework.

The next significant output of this work is the decision about the most suitable real time operating system for the CoSMic platform. The system Xenomai was chosen, because of its flexibility and especially its clear and reasonable future plans.

The Haptic Module was implemented to safely exchange data between the *Sensable Phantom Desktop* haptic device and the framework. It also provides a possibility of the axis locking. The module can be asked to use the force generating capabilities of a haptic device to prevent a handle movement in a direction of a certain axis. Moreover, this module provides a software finite state machine to enable behavior switching on the fly, and to enable a smooth and safe initialization of the device. All these capabilities were successfully experimentally verified. The haptic device was used to control three actuators and another software module was commanding the device to generate desired forces.

The Machine Vision Module was implemented especially to provide a camera independent envelop to hold a user defined algorithm. This envelop was supposed to be real time capable, but this requirement was not met completely due to given restrictions. The main restriction is the absence of a suitable FireWire driver compatible at least with one of the discussed real time operating systems. However, care was taken to let this restriction to be the only one preventing the module from real time run. This means to implement several measures, especially concerning a

pre-allocations of all data as discussed in Section 4.6.

The module is designed as a chain where each part represents one responsibility and runs in its own thread. The first part is responsible for the image acquisition. It is based on tailored parts of OpenCV framework, because it provides support for a vast range of camera types. The original OpenCV code does not provide all needed functionality, so the code had to be extended. Since the module is intended to be used together with FireWire cameras, the most of the extensions are FireWire related. However, the code is written that way, that it is possible to add the extended features for another camera types easily, but if a user does not need them, it is not necessary to implement them.

The next part of the chain is the image processing. In fact, this is the envelop to hold the user defined image processing algorithm. A compromise between the effort of the developer and the effort of a user was chosen. The user has to implement his algorithm in C++ using the provided base class, which takes care of all needed background operations as data exchange between the threads etc. A user need not care about the chain implementation details, but he still has to create his own C++ class.

The whole chain uses classes from OpenCV to represent image data, so it is suggested to base the custom algorithm on this framework, although it is not necessary.

The chain contains also a Graphical User Interface (GUI) responsible for the data visualization and communication with a user. A widget to display OpenCV images in Qt based GUIs was developed. It is ready to be used in any other software project without any changes.

A generic single writer multiple reader circular buffer class was developed to exchange data among threads. Since the class is based on a template, it is also ready to be used in another project immediately.

Several tests were taken to determine module's qualities. The first experiment is based on the code profiling and it showed that the image format conversion could be surprisingly CPU time expensive. It is very important to use a properly configured

camera and thus avoid such conversions when possible.

The second experiment successfully verified the memory pre-allocation measures. The memory pre-allocation is a prerequisite for the desired real time behavior.

The next experiment proved that the real time operation system Xenomai was still able to achieve smaller maximal image processing times, despite the discussed obstacles the module contains. However, the test showed that the non real time solution was more suitable when the lower average processing times were more important than the maximal ones.

The last experiment was meant to show possible data losses due to buffer overflows. It proved there were no losses between the image acquisition part and the image processing part. This result is very important, because any image data loss between those two parts is unacceptable. Certain data loss between the processing and the GUI can be tolerated. This experiment showed it was very important to adjust the sampling timer the GUI uses to grab data from the output buffer to a current frame rate in order to lover the the losses in this spot.

6.2 Future Work

The next step that must be taken is to integrate both of the modules into the CoSMic framework. The integration was not completed, because the needed specifications were not ready at the time of development of the modules.

A task of a high importance will probably be to provide a better support for the custom image processing algorithms definition. For example, to create a tool converting existing image processing scripts in the Matlab language into dynamic libraries attachable to the module.

The last of the most serious topics is the need to create a new FireWire controller driver compatible with Xenomai and thus capable of the real time behavior.

Appropriate effort should be invested in performance enhancements. First logical step on this way is to use the IPP to possibly speed up OpenCV operations. A good next step would be to ensure working OpenGL on a targeted computer. That usually

means to make proprietary drivers run together with the Xenomai kernel, or to use a GPU with sufficient support in the Linux kernel - typically Intel integrated GPUs.

The source code documentation has a special section called ”todo”. It is a list of recommended minor changes to improve the code.

There is a place for different tasks than the changes in the source code. I recommend to take more tests with the Machine Vision module. Especially tests to determine how much is the performance dependent on thread priorities and tests to determine suitable lengths of the buffers. Then the described processing times test should be enhanced by adding a a histogram creation functionality.

BIBLIOGRAPHY

[1] ESSEN, Mathias von. *Control Software for Micro Robotic Platform*. Tampere, 2010. 75 p. Master of Science Thesis. Tampere University of Technology.

[2] SALISBURY, Kenneth. *Haptics: The Technology of Touch* HPCwire Special. Nov. 10, 1995. Van Dam, Andries. "Post-Wimp User Interfaces: The Human Connection." Available at: `http://www.sensable.com/documents/documents/Salisbury_Haptics95.pdf`

[3] *PHANTOM Desktop Haptic Device* [online]. 2010 [cited 2010-07-08]. SensAble Products and services. Available at: `http://www.sensable.com/haptic-phantom-desktop.htm`

[4] VAN HEESCH, Dimitri. *Doxygen* [online]. 1997-10-27, 2010-06-15 [cited 2010-06-23]. Generate documentation at source code. Available at: `http://www.doxygen.nl/index.html`.

[5] MCKENNEY, Paul. *"Real Time" vs. "Real Fast": How to Choose?*. Ottawa Linux Symposium [online]. 2008-06-23, [cited 2010-07-08]. Available at: `http://www.rdrop.com/users/paulmck/realtime/paper/RealTimeVsRealFast.2008.07.23a.pdf`.

[6] *Real-time operating system* [online]. 2006, [cited 2010-06-23]. Wikipedia. Available at: `http://en.wikipedia.org/wiki/Real_time_operating_system`.

[7] KUČERA, Pavel. *Introduction to Real Time Operation Systems*[online, Czech language]. 2008, [cited 2010-06-23]. Available at: `http://sciotech.cz/tc/lectures/mrts/data/01_uvod_cz.pdf`

[8] *System Management Mode* [online]. 2010, [cited 2010-06-23]. Wikipedia. Available at: `http://en.wikipedia.org/wiki/System_Management_Mode#Entering_SMM`

[9] *Linux Kernel Configuration Context Help.* [cited 2010-07-12]. CONFIG_PREEMPT option.

[10] *How does the CONFIG_PREEMPT_RT patch work* [online]. 2001, 2010-04-01 [cited 2010-07-08]. Real-Time Linux Wiki. Available at: `https://rt.wiki.kernel.org/index.php/Frequently_Asked_Questions#How_does_the_CONFIG_PREEMPT_RT_patch_work.3F`

[11] YAGHMOUR, Karim. *Adaptive Domain Environment for Operating Systems.* 2001, Available at: `http://opersys.com/ftp/pub/Adeos/adeos.pdf`

[12] *Life With Adeos.* [online], 2005, [cited 2010-07-08]. Available at: `http://www.xenomai.org/documentation/xenomai-2.0/pdf/Life-with-Adeos.pdf`

[13] *A Tour of the Native API.* [online], 2006, [cited 2010-07-08]. Available at: `http://www.xenomai.org/documentation/branches/v2.3.x/pdf/Native-API-Tour-rev-C.pdf`

[14] *How can GDB be used?.* [online], 2010, [cited 2010-07-08]. Xenomai FAQ. Available at: `http://www.xenomai.org/index.php/FAQs#How_can_GDB_be_used.3F`

[15] *Real-Time Driver Model.* [online], 2010, [cited 2010-07-08]. Available at: `http://www.xenomai.org/documentation/branches/v2.4.x/html/api/group__rtdm.html`

[16] MANDUCHI, Gabriele, BARBALACE, Antonio, *RTAI: Embedded Linux vs Legacy RTOS.* 2007, Available at: `http://www.dei.unipd.it/corsi/so2/RTAI/RTAI.pdf`

[17] *RTAI Configuration Context Help.* [cited 2010-07-12], CONFIG_RTAI_IMMEDIATE_LINUX_SYSCALL option.

[18] *RTAI-Lab project.* [online], 2006-01-31, 2006-07-13, [cited 2010-07-12], RTAI - the RealTime Application Interface for Linux from DIAPM, Available

at: https://www.rtai.org/index.php?module=pagemaster&PAGE_user_op=view_page&PAGE_id=8

[19] *Linux Control and Measurement device interface* [online], 2006, [cited 2010-07-12], Available at: http://www.comedi.org/

[20] *What are the differences between Xenomai and RTAI?*. [online], 2010, [cited 2010-07-08]. Xenomai FAQ. Available at: http://www.xenomai.org/index.php/FAQs#What_are_the_differences_between_Xenomai_and_RTAI.3F

[21] *Embedded Device Support* [online]. 2006, [cited 2010-07-08]. Xenomai. Available at: http://www.xenomai.org/index.php/Embedded_Device_Support

[22] *Towards Xenomai 3* [online]. 2006, [cited 2010-07-08]. Xenomai:Roadmap. Available at: http://www.xenomai.org/index.php/Xenomai:Roadmap#Toward_Xenomai_3

[23] *Qt framework* [online], 2008, [cited 2010-07-08]. Available at: http://qt.nokia.com/products

[24] *QtCreatorWhitepaper* [online]. 2010 [cited 2010-07-08]. Qt Developer Network. Available at: http://developer.qt.nokia.com/wiki/QtCreatorWhitepaper

[25] *Signals and Slots* [online]. 2010 [cited 2010-07-08]. Qt 4.6. Available at: http://doc.trolltech.com/4.6/signalsandslots.html

[26] *Why Doesn't Qt Use Templates for Signals and Slots?* [online]. 2010 [cited 2010-07-08]. Qt 4.6. Available at: http://doc.trolltech.com/4.6/templates.html

[27] *Meta-Object System* [online]. 2010 [cited 2010-07-08]. Qt 4.6. Available at: http://doc.trolltech.com/4.6/metaobjects.html

[28] *Using the Meta-Object Compiler (moc)* [online]. 2010 [cited 2010-07-08]. Qt 4.6. Available at: http://doc.trolltech.com/4.6/moc.html

[29] *OpenCV Wiki* [online]. 2010, 2010-06-10 [cited 2010-06-23]. Welcome. Available at: http://opencv.willowgarage.com/wiki/.

[30] *Intel® Software Network*, [cited 2010-06-23]. Available at: http://software.intel.com/en-us/intel-ipp/

[31] *Real-Time Workshop® 7*, [cited 2010-06-23]. Available at: http://www.mathworks.com/mason/tag/proxy.html?dataid=9547&fileid=43808

[32] *Linux Graphics Essay* [online]. 2009-01-25, [cited 2010-06-23]. The Linux Foundation. Available at: http://www.linuxfoundation.org/collaborate/workgroups/technical-advisory-board-tab/linuxgraphicsessay

[33] *Xenomai API* [online]. 2010, 2010-05-05 [cited 2010-06-23]. Dynamic memory allocation services. Available at: http://www.xenomai.org/documentation/xenomai-2.5/html/api/group__heap.html.

[34] *Profiling* [online]. 2001, 2010-05-26 [cited 2010-06-29]. Wikipedia. Available at: http://en.wikipedia.org/wiki/Profiling_%28computer_programming%29

[35] *Callgrind: a call-graph generating cache profiler* [online]. 2009, [cited 2010-06-29]. Valgrind User Manual. Available at: http://valgrind.org/docs/manual/cl-manual.html

[36] *The KCachegrind Handbook* [online]. 2009-10-07, [cited 2010-06-29]. KDE documentation. Available at: http://docs.kde.org/stable/en/kdesdk/kcachegrind/index.html

[37] *Valgrind* [online]. 2009, [cited 2010-06-29]. Valgrind Home. Available at: http://valgrind.org/

[38] *Massif: a heap profiler* [online]. 2009, [cited 2010-06-29]. Valgrind User Manual. Available at: http://valgrind.org/docs/manual/ms-manual.html

LIST OF SYMBOLS, PHYSICAL CONSTANTS AND ABBREVIATIONS

API	Application Program Interface
CPU	Central Processing Unit
CoSMic	Control Software for Microrobotic Platform
GPU	Graphical Processing Unit
GUI	Graphical User Interface
IDE	Integrated Development Environment
IPP	Intel Integrated Performance Primitives
OpenGL	Open Graphics Library
POSIX	Portable Operating System Interface
RTDM	Real-Time Driver Model
RTOS	Real Time Operating System
RT	Real Time
SMI	System Management Interrupt
SMM	System Management Mode
STL	Standard Template Library
ViCo	Vision System Control

A HOW TO COMPILE LINUX KERNEL AS QUICK AS POSSIBLE

To set Xenomai of RTAI up and running means to compile patched kernel. Kernel compilation is well known for its high time consumption. The purpose of this text it to provide several hints on how to significantly speed up the compilation process. A prerequisite of this text is a basic knowledge of the kernel compilation process on *Debian* like Linux systems (you just need to know the way of compiling a kernel on *Ubuntu*).

A.1 Config File

The easiest way to compile a working custom kernel is to use the *config* file of your currently running distribution kernel (located in `/boot`) as a basic configuration file for the custom kernel. On the other hand, to use this file leads to a very long compilation time, because a distribution kernel has to support vast variety of devices etc. So the very first hint is: Make your kernel configuration file as minimal as possible.

A.2 distcc

The next hint is a little bit more sophisticated: Use distributed compilation. Usually there are several computers in a local network and it is possible to abuse them for the compilation. There is a Linux tool *distcc* providing possibility of distributed compilation.

A.3 ccache

The last hint is also based on usage of one handy tool - *ccache*. This tool creates in fact a compiler cache, so it can speed up recompilations. This is useful, because

usually at least few recompilations are need.

A.4 All Together Step by Step

The very last step is to put all those hints together. Let us call the main computer *master* and the helper computers in the network *slaves*. Moreover, let us assume, that all the computers contain typical development tools like *gcc* already as well as the described tools.

Steps to be taken on every slave:

1. `export PATH="/usr/lib/ccache:$PATH"`
2. `distccd -daemon -allow 192.168.99.12`

The IP adress is the address of the master.

Steps to be taken on the master:

1. `export CONCURRENCY_LEVEL=10`
2. `export PATH="/usr/lib/ccache:$PATH"`
3. `export DISTCC_HOSTS="130.230.42.65 localhost"`
4. `sudo MAKEFLAGS="HOSTCC=/usr/bin/gcc CCACHE_PREFIX=distcc" make-kpkg -rootcmd fakeroot -initrd kernel-image`

The `CONCURRENCY_LEVEL` determines how many jobs will be the compilation split into. Adjust it according to the overall number of processors (I usually use $2 \cdot N$, where N is the number of CPUs). The `DISTCC_HOSTS` is a list of slave IP addresses, separated with spaces - do not forget to include `localhost`.

The result of all this effort should be a *deb* package with the new kernel and it should be built very quickly.

This procedure can be easily tailored to any other time consuming compilation.

B SOFTWARE VERSIONS

Table B.1 shows versions of all used or discussed software (if it is possible to determine a version).

Tab. B.1: Versions of used software

	version
Linux kernel	2.6.31
Xenomai	2.5.3
RTAI	3.8
Qt	4.6.3
Qt Creator	1.3.1
OpenCV	2.1.0